TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

computer languages

Institut für Computersprachen
AB Programmiersprachen und Übersetzer

15. Kolloquium

Programmiersprachen und Grundlagen der Programmierung

Maria Taferl

12.-14. Oktober 2009

Jens Knoop, Adrian Prantl (Hrsg.)

Jens Knoop
Adrian Prantl
Institut für Computersprachen

Technische Universität Wien
http://www.complang.tuwien.ac.at

Bericht 2009-X-1
Schriftenreihe des Instituts für Computersprachen
Technische Universität Wien

Vorwort

Das Kolloquium *Programmiersprachen und Grundlagen der Programmierung* (*KPS*) findet dieses Jahr zum 15. Mal statt. Es setzt eine Reihe von Arbeitstagungen fort, die ursprünglich von den Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel) ins Leben gerufen wurde.

Aus den ursprünglich drei Arbeitsgruppen sind in der Zwischenzeit weitere Forschungsgruppen in ganz Deutschland und darüberhinaus hervorgegangen. Seit einigen Jahren präsentiert sich die Veranstaltung als ein offenes Forum für interessierte deutschsprachige Wissenschafter. Die folgende Liste gibt einen Überblick über die bisherigen Tagungsorte und Veranstalter und zeigt die lange Tradition der KPS-Treffen:

1980	Tannenfelde im Aukrug	Universität Kiel
1982	Altenahr	RWTH Aachen
1985	Passau	Universität Passau
1987	Midlum auf Föhr	Universität Kiel
1989	Hirschegg	Universität Augsburg
1991	Rothenberge bei Steinfurth	Universität Münster
1993	Garmisch-Partenkirchen	Universität der Bundeswehr München
1995	Alt-Reichenau	Universität Passau
1997	Avendorf auf Fehmarn	Universität Kiel
1999	Kirchhundem-Heinsberg	FernUniversität in Hagen
2001	Rurberg in der Eifel	RWTH Aachen
2004	Freiburg-Munzingen	Universität Freiburg
2005	Fischbachau	LMU München
2007	Timmendorfer Strand	Universität Lübeck

Das 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2009) wird vom Institut für Computersprachen der Technischen Universität Wien organisiert. Wir freuen uns, zu diesem Jubiläumstreffen mehr als 50 Teilnehmer von 28 Universitäten aus Österreich, der Schweiz, Deutschland, Holland, England und Kanada vom 12. bis 14. Oktober 2009 im Wallfahrtsort Maria Taferl am Österreichischen Jakobsweg begrüßen zu können. Besonders freuen wir uns, unter den Teilnehmern Prof. Dr. Dr.h.c. Hans Langmaack als einen der Gründerväter dieser Tagungsreihe begrüßen zu können, sowie Prof. Dr. Rudolf Berghammer, Prof. Dr. Peter Thiemann, Prof. Dr. Clemens Grelck und Dr. Annette Stümpel als Organisatoren früherer KPS-Treffen. Ganz besonders freuen wir uns, dass nahezu alle Teilnehmer am diesjährigen KPS-Treffen in einem Vortrag über ihre Forschungsarbeit berichten werden. Um dies im Rahmen der zur Verfügung stehenden Zeit zu ermöglichen, verzichten wir in diesem Jahr auf einen eingeladenen Hauptvortrag.

Der vorliegende Tagungsband ist als Bericht 2009-X-1 des Instituts für Computersprachen der TU Wien erschienen. Er enthält 46 Beiträge, zum Teil in Form von Kurzzusammenfassungen, von 62 Autoren. Weitere spät eingegangene Beiträge sind in einem Ergänzungsband gesammelt, der als Bericht 2009-X-2 des Instituts für Computersprachen der TU Wien erscheint. Alle Beiträge aus Haupt- und Ergänzungsband zeigen die Breite der wissenschaftlichen Forschung im Bereich Programmiersprachen und Grundlagen der Programmierung im deutschsprachigen Raum. Eine CD-ROM fasst zusätzlich alle Beiträge der beiden Tagungsbände zusammen. Unser Dank gilt allen Autoren für ihren Einsatz und die gute Zusammenarbeit, die diese beiden Bände ermöglicht haben. Unser besonderer Dank gilt darüberhinaus Frau Ewa Vesely und Herrn Leonid Narinsky, die bei der Vorbereitung und Organisation dieses Treffens von der Erstellung der Webseite über die Planung der Exkursion bis zum schnellen Auftun zusätzlicher Unterbringungsmöglichkeiten nach dem Hinauslaufen aus der Kapazität des Tagungshauses Hervorragendes geleistet haben. Unseren besonderen Dank drücken

wir dabei auch allen Mitarbeitern des Hauses Hotel Schachner für die gute Zusammenarbeit bei der Planung dieses Treffens aus.

Wir wünschen allen Teilnehmern interessante und spannende Vorträge, fruchtbare Diskussionen und Anregungen für die Forschungsarbeit, das Kennenlernen neuer Kollegen und das Wiedersehen guter Bekannter, das Anbahnen und Knüpfen neuer Kooperationen und die Verbreiterung und Vertiefung bestehender, eine erlebnis- und abwechslungsreiche Exkursion zu dem zum UNESCO-Weltkulturerbe gehörenden Benediktinerklosters Stift Melk und einen angenehmen und anregenden Aufenthalt in Maria Taferl am Eingang zur Wachau.

Willkommen zur KPS 2009!

Wien, im Oktober 2009

Jens Knoop
Adrian Prantl

Teilnehmer

Sebastian Altmeyer	Universität des Saarlandes, Saarbrücken
Wolfram Amme	Friedrich-Schiller-Universität Jena
Enes Bajrović	Universität Wien
Gergö Barany	Technische Universität Wien
Rudolf Berghammer	Christian-Albrechts-Universität zu Kiel
Dirk Beyer	Simon Fraser University, Surrey, B.C.
Walter Binder	Università della Svizzera italiana, Lugano
Florian Brandner	Technische Universität Wien
Stefan Brunthaler	Technische Universität Wien
Jens Dörre	Universität Passau
M. Anton Ertl	Technische Universität Wien
Christoph Feller	Technische Universität Kaiserslautern
Sebastian Fischer	Christian-Albrechts-Universität zu Kiel
Gerhard Goos	Universität Karlsruhe (TH)
Clemens Grelck	Universiteit van Amsterdam und University of Hertfordshire
Jürg Gutknecht	ETH Zürich
Sebastian Hack	Universität des Saarlandes, Saarbrücken
Michael Hanus	Christian-Albrechts-Universität zu Kiel
Thomas Heinze	Friedrich-Schiller-Universität Jena
Christoph Höger	Technische Universität Berlin
Christina Jansen	RWTH Aachen
Tudor Jebelean	RISC, Johannes Kepler Universität Linz
Raimund Kirner	Technische Universität Wien
Jens Knoop	Technische Universität Wien
Peter Lammich	Westfälische Wilhelms-Universität Münster
Oliver Lampl	Alpen-Adria-Universität Klagenfurt
Hans Langmaack	Christian-Albrechts-Universität zu Kiel
Florian Lorenzen	Technische Universität Berlin
Tim A. Majchrzak	Westfälische Wilhelms-Universität Münster
Volker Mattick	Technische Universität Dortmund
Eduard Mehofer	Universität Wien
Thomas Meyer	Universität Basel
Patrick Michel	Technische Universität Kaiserslautern
Leonid Narinsky	Technische Universität Wien
Ulrich Neumerkel	Technische Universität Wien
Nikolaj Popov	RISC, Johannes Kepler Universität Linz
Adrian Prantl	Technische Universität Wien
Franz Puntigam	Technische Universität Wien
Fabian Reck	Christian-Albrechts-Universität zu Kiel
Dirk Richter	Martin-Luther-Universität Halle-Wittenberg

Wolfgang Scholz	Universität Passau
Sven-Bodo Scholz	University of Hertfordshire
Markus Schordan	Fachhochschule Technikum Wien
Dietmar Schreiner	Technische Universität Wien
Martin Schwarz	Technische Universität München
Lukas Stadler	Johannes Kepler Universität Linz
Annette Stümpel	Universität zu Lübeck
Michael Tautschnig	Technische Universität Darmstadt
Peter Thiemann	Universität Freiburg
Baltasar Trancón y Widemann	Universität Bayreuth
Christian Tschudin	Universität Basel
Helmut Veith	Technische Universität Darmstadt
Alexander Wenner	Westfälische Wilhelms-Universität Münster
Thomas Würthinger	Johannes Kepler Universität Linz
Wolf Zimmermann	Martin-Luther-Universität Halle-Wittenberg
Florian Zuleger	Technische Universität Darmstadt

Inhaltsverzeichnis

Static Timing Analysis for Hard Real-Time Systems
von *Sebastian Altmeyer, Mohamed Abdel Maksoud, Claire Burguiere, Daniel Grund, Jörg Herter, Philipp Lucas, Oleg Parshin, Markus Pister, Jan Reineke, Marc Schlickling, Björn Wachter und Reinhard Wilhelm* 1

Erfahrungen bei der Auswahl von maschinenunabhängigen Optimierungen im Bereich des mobilen Codes
von *Wolfram Amme* 13

Programming Support for Cell/BE Multiprocessor
von *Enes Bajrović und Eduard Mehofer* 17

SATIrE within ALL-TIMES: Improving Timing Technology with Source Code Analysis
von *Gergö Barany* 27

Computing and Visualizing Closure Objects Using Relation Algebra and RelView
von *Rudolf Berghammer und Bernd Braßel* 38

Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine
von *Walter Binder, Alex Villazón, Danilo Ansaloni und Philippe Moret* 51

Automatic Tool Generation from Structural Processor Descriptions
von *Florian Brandner* 65

Inline Caching meets Quickening
von *Stefan Brunthaler* 67

Typsicheres und generisches MapReduce
von *Jens Dörre* 68

Utilizing Multiple Hardware Threads with Pipeline Parallelism
von *M. Anton Ertl* 69

A Sound, Complete and Usable Hoare-Style Logic for a Sequential Java Subset
von *Christoph Feller* 76

Reinventing Haskell Backtracking
von *Sebastian Fischer* 77

Concurrency Engineering with S-Net
von *Clemens Grelck, Sven-Bodo Scholz und Alex Shafarenko* 78

Automatische Paketisierung
von *Sebastian Hack* 93

Set Functions for Functional Logic Programming
von *Michael Hanus* 94

Verbesserung der Modellbildung und Analyse verteilter Geschäftsprozesse durch Prozessumstrukturierung
von *Thomas Heinze, Wolfram Amme und Simon Moser* 95

Generation of Incremental Parsers for Modern IDEs
von *Christoph Höger* ... 104

Generierung von Hyperkantenersetzungsgrammatiken zur Heapabstraktion
von *Christina Jansen und Jonathan Heinen* ... 117

Combining Automated Reasoning and Algebraic Methods in Theorema
von *Tudor Jebelean* ... 124

Automatic Calculation of Coverage Profiles for Coverage-based Testing
von *Raimund Kirner und Walter Haas* ... 126

On Undecidability Results of Real Programming Languages
von *Raimund Kirner, Wolf Zimmermann und Dirk Richter* ... 141

From Trusted Annotations to Verified Knowledge
von *Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec und Markus Schordan* ... 155

Tree Automata for Analyzing Dynamic Pushdown Networks
von *Peter Lammich* ... 167

MultiMediaC# – QoS-Aware Programming with C#
von *Oliver Lampl und Laszlo Böszörmenyi* ... 178

Technische Aspekte des erfolgreichen Testens von Software in Unternehmen
von *Tim A. Majchrzak* ... 193

An Algebraic Framework for Modeling and Processing Hyperdocuments
von *Volker Mattick* ... 208

Maintaining XML Data Integrity in Programs – An Abstract Datatype Approach
von *Patrick Michel* ... 219

Lambdas und Schleifen in monotonen Logikprogrammen
von *Ulrich Neumerkel* ... 220

Functional Program Verification in Theorema. Soundness and Completeness
von *Nikolaj Popov und Tudor Jebelean* ... 221

Towards a Static Profiler
von *Adrian Prantl* ... 230

How to Specify the Flow of Data Accessibility: An OO Way of Concurrent Programming
von *Franz Puntigam* ... 231

Towards a Parallel Search for Solutions of Non-deterministic Computations
von *Fabian Reck und Sebastian Fischer* ... 243

Rekursionspräzise Intervallanalysen
von *Dirk Richter* ... 244

A New Look on Data Parallelism: Space vs. Time
von *Sven-Bodo Scholz* ... 259

Statische Erkennung von semantischen Feature-Interaktionen
von *Wolfgang Scholz* ... 260

ARAL: a Language for Information Exchange between Program Analysis Tools
von *Markus Schordan* 261

Robots, Software, Mayhem? Towards a Design Methodology for Robotic Software Systems
von *Dietmar Schreiner* 262

A Formalisation of the OSEK Concurrency Model
von *Martin Schwarz* 263

Lazy Continuations for Java Virtual Machines
von *Lukas Stadler* 264

Communication-based System Development using Standard Programming Languages
von *Annette Stümpel* 265

FQL: A Query Language for Program Testing
von *Andreas Holzer, Christian Schallhart, Michael Tautschnig und Helmut Veith* 269

Type-Safe Bytecode Generation in Scala
von *Peter Thiemann* 284

Programmierung als Leitbild in der Theorie der Ökosysteme
von *Baltasar Trancón y Widemann* 285

Programming by Equilibria
von *Christian Tschudin und Thomas Meyer* 286

Adding Weights to Dynamic Pushdown Networks
von *Alexander Wenner* 287

Towards Dynamic Code Evolution for the HotSpot VM
von *Thomas Würthinger* 289

Static Timing Analysis for Hard Real-Time Systems

Sebastian Altmeyer, Mohamed Abdel Maksoud, Claire Burguiere, Daniel Grund, Jörg Herter, Philipp Lucas, Oleg Parshin, Markus Pister, Jan Reineke, Marc Schlickling, Björn Wachter, Reinhard Wilhelm

Compiler Design Lab, Saarland University

Abstract. Hard real-time systems impose strict timing constraints. To prove that these constraints are met, timing analyses aim to derive safe upper bounds on a task's execution time. Especially modern processor features (caches, out-of-order pipelines, etc.) have a strong impact on the variation of a task's execution time and thus, on the precision and the complexity of the timing analysis. Naive or measurement-based approaches usually can not guarantee safe and reliable timing bounds.
This paper provides an overview of the current timing analysis technique and shortly present ongoing research at the Compiler Design Lab at Saarland University. An extended version can be found in [Wil05].

1 Static Timing Analysis

Any software system when executed on a modern high-performance processor shows a certain variation in execution time depending on the input data, the initial hardware state, and the interference with the environment. This article treats timing analysis of tasks with uninterrupted execution. In general, the state space of input data and initial states is too large to exhaustively explore all possible executions in order to determine the exact worst-case and best-case execution times. Instead, bounds for the execution times of basic blocks are determined, from which bounds for the whole system's execution time are derived. Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions lose information, and thus are in part responsible for the gap between WCETs and upper bounds and between BCETs and lower bounds. How much is lost depends both on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software. The methods used to determine upper bounds and lower bounds are very similar.

In modern microprocessor architectures, caches, pipelines, and all kinds of speculation are key features for improving (average-case) performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution time of individual instructions, and thus the contribution to the program's execution time can

vary widely. The interval of execution times for one instruction is bounded by the execution times of the following two cases:

- The instruction goes "smoothly" through the pipeline; all loads hit the cache, no pipeline hazard happens, i.e., all operands are ready, no resource conflicts with other currently executing instructions exist.
- "Everything goes wrong", i.e., instruction and/or operand fetches miss the cache, resources needed by the instruction are occupied, etc.

We will call any increase in execution time during an instruction's execution a *timing accident* and the number of cycles by which it increases the *timing penalty* of this accident. Timing penalties for an instruction can add up to several hundred processor cycles. Whether the execution of an instruction encounters a timing accident depends on the execution state, e.g., the contents of the cache(s), the occupancy of other resources, and thus on the execution history. It is therefore obvious that the attempt to predict or exclude timing accidents needs information about the execution history.

1.1 Timing Analysis Framework

Over the last several years, a more or less standard architecture for timing-analysis tools has emerged [HWH95,TFW00,Erm03]. 1 gives a general view on this architecture. First, one can distinguish three major building blocks:

- Control-flow reconstruction and static analyses for control and data flow.
- Micro-architectural analysis, which computes upper and lower bounds on execution times of basic blocks.
- Global bound analysis, which computes upper and lower bounds for the whole program.

The following list presents the individual phases and describes their objectives and problems. Note that the first four phases are part of the first building block.

1. *Control-flow reconstruction* [The02a] takes a binary executable to be analyzed, reconstructs the program's control flow and transforms the program into a suitable intermediate representation. Problems encountered are dynamically computed control-flow successors, e.g. stemming from switch statements, function pointers, etc..
2. *Value analysis* [CC77,SS07] computes an over-approximation of the set of possible values in registers and memory locations by an interval analysis and/or congruence analysis. This information is among others used for a precise data-cache analysis.
3. *Loop bound analysis* [EG97,HSR$^+$00] identifies loops in the program and tries to determine bounds on the number of loop iterations, information indispensable to bound the execution time. Problems are the analysis of arithmetic on loop counters and loop exit conditions, as well as dependencies in nested loops.

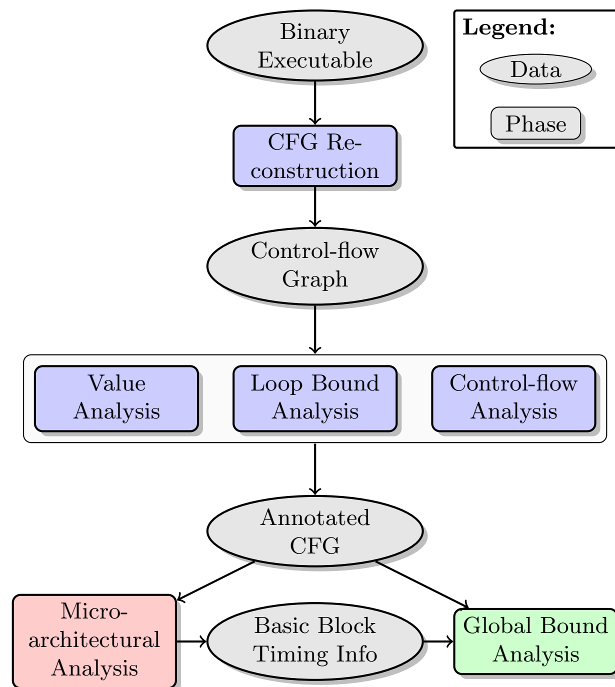


Fig. 1. Main components of a timing-analysis framework and their interaction.

4. *Control-flow analysis* [EG97,SM07a] narrows down the set of possible paths through the program by eliminating infeasible paths or to determine correlations between the number of executions of different blocks using the results of value analysis results. These constraints will tighten the obtained timing bounds.
5. *Micro-architectural analysis* [Eng02,The04,FW99] determines bounds on the execution time of basic blocks by performing an abstract interpretation of the program, taking into account the processor's pipeline, caches, and speculation concepts. Static cache analyses determine safe approximations to the contents of caches at each program point. Pipeline analysis analyzes how instructions pass through the pipeline accounting for occupancy of shared resources like queues, functional units, etc.. Ignoring these average-case-enhancing features would result in imprecise bounds.
6. *Global bound analysis* [LM95,The02b] finally determines bounds on execution time for the whole program. Information about the execution time of basic blocks is combined to compute shortest and longest paths through the program. This phase takes into account information provided by the loop bound- and control-flow analysis.

The commercially available tool `aiT` by AbsInt, cf. `http://www.absint.de/wcet.htm`, implements this architecture. It is used in the aeronautics and automotive industries and has been successfully used to determine precise bounds on execution times of real-time programs [FW99,FHL+01,TSH+03,HLTW03].

1.2 Timing Anomalies

Most powerful microprocessors have so-called *timing anomalies.* Timing anomalies are contra-intuitive influences of the (local) execution time of one instruction on the (global) execution time of the whole program. Several processor features can interact in such a way that a locally faster execution of an instruction can lead to a globally longer execution time of the whole program.

For example, a cache miss contributes the cache-miss penalty to the execution time of a program. It was, however, observed for the MCF 5307 [RSW02], that a cache miss may actually speed up program execution. Since the MCF 5307 has a unified cache and the fetch and execute pipelines are independent, the following can happen: A data access that is a cache hit is served directly from the cache. At the same time, the pipeline fetches another instruction block from main memory, performing branch prediction and replacing two lines of *data* in the cache. These may be reused later on and cause two misses. If the data access was a cache miss, the instruction fetch pipeline may not have fetched those two lines, because the execution pipeline may have resolved a misprediction before those lines were fetched.

2 Work In Progress

In this section, we shortly present recent or ongoing work from our group. This research either aims at reducing the shortcomings of the timing analysis or try to broaden its applicability.

2.1 Constraints on the Control Flow Graph

There are three influences on the execution time of a task:

1. *Task inputs* and *logical state* determine the dynamical sequence of instructions and memory accesses;
2. the *dynamic hardware state* (cache contents, pipeline states) determines how instructions are executed;
3. *static hardware properties* such as length of pipelines and speed of execution units determine the actual execution time.

Whereas before we have considered how to cope with uncertainty in (2) and how to model (3), we now consider ways to improve precision of WCET calculations by restricting the uncertainty arising from (1). We briefly sketch the general ideas; for a deeper explanation, see [WLP$^+$10].

A WCET analysis tool has to consider the worst-case path through the control-flow graph. There are various ways of conceptually restricting the set of possible paths and of expressing these restrictions.

Model-based flow constraints On the one hand, the WCET tool might regard a path as feasible, and thus potentially a worst-case path, which actually is infeasible[1]. Although there is work on detecting infeasible paths in the executable to be analysed (e.g., [SM07b]), the scope of the analysis is only a *single* run through the task. Correlations between control flow choices such as `a>0` and `a<=0` may be detected in this way. However, correlations between variables which arise from the execution of a task in a loop (*inter-run* properties) cannot be determined by the analysis.

For example, if control logic is implemented by automata in a high-level design tool such as Stateflow/Simulink, not all combinations of automata states are reachable. This restriction on control state in turn restricts the possible behaviours of the model, which translates to restrictions on the control flow path through the model's implementation. Analysis of Stateflow automata and their effect on the Simulink model they control thus can lead to control flow restrictions. These restrictions may be expressed to the WCET tool as *flow constraints* or generalisations thereof: Flow constraints introduce additional inequalities relating the execution counts of basic blocks in the ILP which determines the actual worst-case path.

[1] That is, it cannot happen during any execution, though it does not contain any dead code.

Mode-specific input constraints Another way of restricting the control flow is by considering *mode-specific* behaviours. Embedded control systems often have radically different *behaviours* depending on their operating mode (e.g., start-up, running, shut-down and error). This concerns for example their WCETs or accesses to shared ressources. Modes also may impose specific *requirements* on the tasks such as mode-specific deadlines. An accurate analysis thus requires the determination of mode-specific WCETs.

Mode-specific WCET analysis strives to infer operating modes from source code, to identify those operating modes with significantly differing WCETs and to compute mode-specific WCETs. To this end, an analysis of the syntactical and semantical properties of a C program heuristically infers operating modes by considering code patterns and mode-signifying differences in behaviour. The determination of mode-specific WCET bounds can then be done by annotations in several ways. Again, flow constraints can be employed to model the *effect* of a mode on the possible control flows. Furthermore, *input constraints* can communicate to the tool the set of input variables giving rise to a particular mode. For example, in one mode a sensor variable may be fixed to values $[0, \infty)$ and in another one (an error mode) to -1. Such annotations are used in the earlier value analysis phase of the system. Therefore they effect the complete WCET analysis and lead to more specific results, in contrast to flow annotations which effect only the latest analysis phase.

2.2 Semi-automatic derivation of Pipeline Analyses from VHDL Models

Currently, the pipeline models are *hand-crafted* by human experts [The04]. Therefore the model creation as well as the implementation of the corresponding pipeline analysis is a very time-consuming and error-prone process.

As modern processors are synthesized out of formal hardware description languages, like *VHDL*, in which their behavior (including the timing) is exactly specified, the timing model could be semi-automatically derived from it. This would avoid errors introduced by manual implementations due to human involvement and it would speed up the process, too.

VHDL models of real world processors are usually very big and complex. Just generating a pipeline analysis that covers the whole micro-architectural behavior[2] would additionally increase the state space. This would render the resulting timing analysis infeasible in terms of space and time consumption.

We want to derive computational feasible timing models out of formal *VHDL* specifications in a semi-automatic way. In a first step, we reduce the size of the model by pruning out all parts that does not contribute to the timing behavior. For example, we don't need information about each step within a multiplier unit. Instead, it suffices to know how many clock cycles an instruction occupies each stage of the multiplier pipeline.

[2] in this case, the timing model would be equivalent to the full *VHDL* model

The pruned model still contains a lot of detailed information about the processor state. But for practical reasons it is impossible to represent all state information in full detail. If we were to exactly record e.g. the contents of all memory cells or registers, the space required for the analysis would be prohibitive. Luckily, in many cases the exact knowledge about these things is not important as far as timing is concerned: an addition always takes the same amount of time, no matter what the arguments are. In other cases, the timing does depend on such information, but we may choose to lose the exact timing knowledge in order to make the analysis more efficient, or even to make it possible at all. One example for this are multiplications on some architectures, which are faster if one argument has many leading zero bits. By not keeping track of the arguments exactly, we have to assume an entire range of execution times for multiplication. The loss in precision is acceptable in this case, as the difference is usually only a few processor cycles and multiplications are rare.

Therefore, the second part of the timing model derivation is the definition of abstractions on the processor state. Abstractions means that we either left out some details of the processor state or we approximate them. An example for an approximation is the replacement of concrete addresses by address intervals. For memory accesses, we do not need to know the exact address. We only need to know the type of memory that is accessed in order to simulate its timing behavior.

Using the methodology of abstract interpretation, one can trade precision of the analysis against efficiency by choosing different processor abstractions and concretization relations between the concrete processor state and the abstract one.

2.3 FIFO Cache Analysis

Precise and efficient analyses have been developed for set-associative caches that employ the least-recently-used (*LRU*) replacement policy [Alt96,FW99]. Generally, research in the field of embedded real-time systems assumes *LRU* replacement. In practice however, other policies like first-in first-out (*FIFO*) or pseudo-*LRU* (*PLRU*) are also commonly used, e.g. in the *Intel XScale*, some *ARM9* and *ARM11*, and the *PowerPC 75x* series.

Two kinds of information can be naturally distinguished in cache analysis: must-information that allows for predicting hits, and may-information that allows for predicting misses. Previous work showed that it is inherently more difficult to obtain may-information for than for *LRU*; see [RGBW07]. A first step towards the analysis of those policies was the general concept of relative competitiveness; see [Rei08]. Depending on the particular policy, however, a cache analysis based on relative competitiveness may be anything from very precise to ineffective.

In [GRG09], we describe a generic policy-independent framework for cache analysis. It allows for cooperation of may- and must-analyses through a minimal interface, which improves their precision.

In addition, we present a may- and a must-analysis for *FIFO* caches. The must-analysis borrows basic ideas from *LRU*-analysis [FW99]. To predict cache hits, it infers upper bounds on cache misses to prove containedness of memory blocks. To predict cache misses, the may-analysis infers lower bounds on cache misses to prove eviction. By taking into account the order in which hits and misses happen, we improve the may-analysis, thereby increasing the number of predicted cache misses. Through the cooperation of the two analyses in the generic framework, this also improves the precision of the must-analysis.

2.4 Branch Target Buffer

In today's systems, caches, deep pipelines, BTBs, and all sorts of speculation are used to increase average-case performance. These features are challenging for timing analysis since they cause a large variability in the execution times of instructions. If an analysis cannot safely exclude spurious detrimental behavior (cache misses, pipeline stalls, etc.) the obtained WCET bounds may become imprecise and thus useless. It depends on the design of the hardware components how well analyses can exclude such behavior.

BTBs cache addresses of branch targets or instructions at branch targets to reduce the latency when processing branches. Branches occur relatively often and the difference in latency between a BTB hit and a miss is large enough to have a significant influence on the execution time. Thus, a BTB analysis is necessary to obtain precise WCET bounds.

We introduce a modular WCET analysis framework for BTBs that can be adapted to various BTB implementations. It consists of a fixed main module that is the same for all BTBs and two parameter modules each of which answers one of the following questions: For which branches does the BTB contain information? What information is stored in the BTB for a given branch? The modules interact via fixed interfaces such that they can be exchanged independently.

Our second contribution is an instantiation of our framework for the *PowerPC*. The *PowerPC* is used in time-critical automotive and avionics systems and features a branch processing unit (BPU) with branch prediction and a BTB. This case study shows the applicability of our framework for a non-trivial case and demonstrates the effort needed to model a hardware feature. This instantiation improves the WCET bounds by on average 13% on a set of avionic and compiler benchmarks. On a subset of the benchmarks measuring execution time was possible, which yields under-approximations of the WCET but allows to bound the overestimation of our analysis. For this subset, our analysis reduced the average overestimation from 54% to 20%.

Our last contribution is the identification of principles of more predictable hardware designs and their influence on WCET bounds. We identify problems regarding predictability of the example BTB we study. Capitalizing on the modularity of our analysis, we propose alternative hardware designs and evaluate them by additional experiments. In case of the *PowerPC*, employing a more predictable replacement policy (*LRU*) in the BTB would improve the computed

WCET bounds by 2.9% and reduce analysis time considerably. Minor modifications that increase uniformity by eliminating special cases would not only simplify analysis but also improve the WCET bounds obtained with our analysis by up to 20%. Finally, we generalize our findings and give advice to hardware designers.

2.5 Heap-allocating Programs

Static worst-case execution time analyses rely on high cache predictability in order to achieve precise bounds on a program's execution time. Such analyses, however, fail to cope with programs using dynamic memory allocation. This is due to the unpredictability of the cache behavior introduced by the dynamic memory allocators.

We investigate two approaches to enable precise worst-case execution time analysis for programs that use dynamic memory allocation.

The first approach automatically transforms the dynamic memory allocation into a static allocation with comparable memory consumption [HR09]. Hence, we try to preserve the main advantage of dynamic memory allocation, namely the reduction of memory consumption achieved by reusing deallocated memory blocks for subsequent allocation requests. However, ending up with a static allocation schemes allows for using existing techniques for WCET analyses.

The second approach replaces the unpredictable dynamic memory allocator by a predictable dynamic allocation [HRW08].

Both approaches rely on precise information about the dynamically allocated heap objects and data structures arising during program executions. Obtaining this information requires an adapted shape analysis together with appropriate abstraction techniques.

2.6 BDDs to improve the Pipeline Analysis

Static timing analysis only becomes computationally feasible in practice by using abstraction, which is applied to both the modeling of processor and program behavior. However, abstraction loses information which leads to uncertainty, e.g. it may not be possible to statically determine the exact address of a memory access. Furthermore, program inputs are not precisely known in advance. At the level of the hardware model, this lack of information is accounted for by non-deterministic choices. To be safe, the analysis has to exhaustively explore all possibilities. This can lead to state explosion making an explicit enumeration of states infeasible due to memory and computation time constraints.

We address the state explosion problem in static WCET analysis by storing and manipulating hardware states in a more efficient data structure based on Ordered Binary Decision Diagrams (BDDs). Our work is inspired by BDD-based symbolic model checking. Symbolic model checking has been successfully applied to components of processors. Its success sparked a general interest in BDDs and other symbolic representations. Today, BDDs are also used extensively to analyze software, e.g. in software model checking and points-to analysis.

We enhance the existing framework for static WCET analysis with a symbolic representation of abstract pipeline models, and assess its effectiveness on a set of industrial benchmarks. We have developed a prototype implementation which is integrated into the commercial WCET analysis tool `aiT`. In our prototype implementation, we employ the model of a real-life processor, the Infineon TriCore. The model was developed and tested within `aiT`. This enables a meaningful performance comparison between the two implementations, which produce the same analysis results.

To arrive at an efficient symbolic analysis that scales to industrial-size programs, we have not only incorporated well-known optimizations from symbolic model checking but also novel domain-specific optimizations that leverage properties of the processor and the program.

2.7 Context Switch Costs

In preemptive real-time systems, scheduling analyses are based on the worst-case response time of tasks. This response time includes upper bounds on the execution time and context switch costs. In case of preemption, cache memories may suffer interferences between memory accesses of the preempted and of the preempting task. These interferences lead to some additional reloads that are referred to as cache-related preemption delay (CRPD). This CRPD constitutes a large part of the context switch costs.

Upper bounds on the CRPD are usually computed using the concept of useful cache blocks (UCB). These are memory blocks that may be in cache before a program point and may be reused after it. When a preemption occurs at that point the number of additional cache-misses is bounded by the number of useful cache blocks. We tighten the CRPD bound by using a modified notion of UCB: Only cache blocks that are definitely cached are considered useful by our approach [AB09].

References

[AB09] Sebastian Altmeyer and Claire Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS '09)*, pages 109–118. IEEE Computer Society, July 2009.

[Alt96] P. Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks.* PhD thesis, Universität Paderborn, 1996.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. pages 238–252, Los Angeles, California, 1977.

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par*, pages 1298–1307, 1997.

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Dept. of Information Technology, Uppsala University, 2002.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, volume 2211 of *LNCS*, pages 469 – 485, 2001.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[GRG09] Daniel Grund, Jan Reineke, and Gernot Gebhard. Branch target buffers: WCET analysis and timing predictability. In *15th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009*, August 2009.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture an the design and the results of WCET tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.

[HR09] Jörg Herter and Jan Reineke. Making dynamic memory allocation static to support WCET analyses. In *Proceedings of 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, June 2009.

[HRW08] Jörg Herter, Jan Reineke, and Reinhard Wilhelm. CAMA: Cache-Aware Memory Allocation for WCET Analysis. In Marco Caccamo, editor, *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, July 2008.

[HSR+00] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *The Journal of Real-Time Systems*, pages 121–148, May 2000.

[HWH95] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. pages 288–297, December 1995.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[Rei08] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, November 2008.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[RSW02] T. Reps, M. Sagiv, and R. Wilhelm. Shape analysis and applications. In Y N Srikant and Priti Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, pages 175 – 217. CRC Press, 2002.

[SM07a] Ingmar Stein and Florian Martin. Analysis of path exclusion at the machine code level. In *WCET*, 2007.

[SM07b] Ingmar Stein and Florian Martin. Analysis of path exclusion at the machine code level. In *Proceedings of the 7th International Workshop on Worst-Case Execution-Time Analysis*, July 2007.

[SS07] Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *MEMOCODE*, pages 39–48, 2007.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[The02a] Henrik Theiling. *Control Flow Graphs For Real-Time Systems Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[The02b] Henrik Theiling. Ilp-based interprocedural path analysis. In *Embedded Software (EMSOFT)*, volume 2491 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2002.

[The04] Stephan Thesing. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[TSH+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, June 2003.

[Wil05] Reinhard Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.

[WLP+10] Reinhard Wilhelm, Philipp Lucas, Oleg Parshin, Lili Tan, and Björn Wachter. Improving the precision of WCET analysis by input constraints and model-derived flow constraints. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*. Springer-Verlag, 2010. To appear.

Erfahrungen bei der Auswahl von maschinenunabhängigen Optimierungen im Bereich des mobilen Codes

Wolfram Amme

Institut für Informatik
Friedrich-Schiller-Universität Jena
Jena, Deutschland
Wolfram.Amme@uni-jena.de

Mobiler Code wird heutzutage auf der Basis einer Just-in-Time (JIT)- Übersetzung ausgeführt. Da die für eine JIT- Übersetzung notwendige Zeit direkt in die Ausführungszeit der Programme einfließt, ist man darauf bedacht, die für eine solche Übersetzung notwendigen Zeiten möglichst gering zu halten. Eine Möglichkeit dieses Vorhaben zu unterstützen, ist schon während der Erzeugung von mobilen Programmen auf der Produzentenseite eines Systems zum Transport mobiler Programme maschinenunabhängige Optimierungen auszuführen.

Im Vortrag wird die Wirkungsweise vorgestellt, die eine Verwendung von maschinenunabhängigen Optimierungen auf die Ausführung von mobilen Programmen hat. Alle dabei betrachteten Experimente wurden unter Verwendung des SafeTSA-Formats durchgeführt, eines auf der SSA-Form basierenden Zwischencodeformats für mobilen Code.

SafeTSA ist die erste absolut referenz- und typensichere auf *Static Single Assignment-Form* basierende Zwischencoderepräsentation für mobilen Code. Die in [1] näher beschriebene Methode wurde als eine Alternative zur JVM und dessen Java-Bytecode entworfen, und hat gegenüber diesen mehrere entscheidene Vorteile: (1) SafeTSA ist besser als Eingabe für einen optimierenden JIT-Compiler geeignet und erlaubt, viele maschinenunabhängige Codeoptimierungen bereits während der Erstellung der Zwischencoderepräsentation auf der Produzentenseite durchzuführen. (2) SafeTSA garantiert die Referenz- und Typensicherheit des gesendeten Programmes durch Konstruktion. Diese Eigenschaft reduziert die zur Sicherheitsgarantie notwendigen Verifikationsarbeiten eines JIT-Compilers auf ein Minimum. (3) SafeTSA besitzt separate Instruktionen zur Nullreferenz- und Arrayindexüberprüfung, damit können überflüssige Nullreferenz- und Arrayindexüberprüfungen schon vor Erzeugung des Zwischencodes eliminiert werden, was zu einem erheblich besseren Laufzeitverhalten von Java-Applikationen führt. (4) Neben diesen Vorteilen ist SafeTSA zusätzlich kompakter als Java-Bytecode.

Zur Überprüfung der Leistungsfähigkeit des von uns geschaffenen SafeTSA-Formats wurde ein vollständiges System zum Transport von mobilen Code entwickelt, mit dem SafeTSA-Programme einerseits generiert bzw. andererseits unter Verwendung einer JIT-Übersetzung ausgeführt werden können. Der prinzipielle Aufbau der für die Programmiersprache Java entworfenen Produzentenseite des Systems ist in Abbildung 1 wiedergegeben. Die Produzentenseite führt

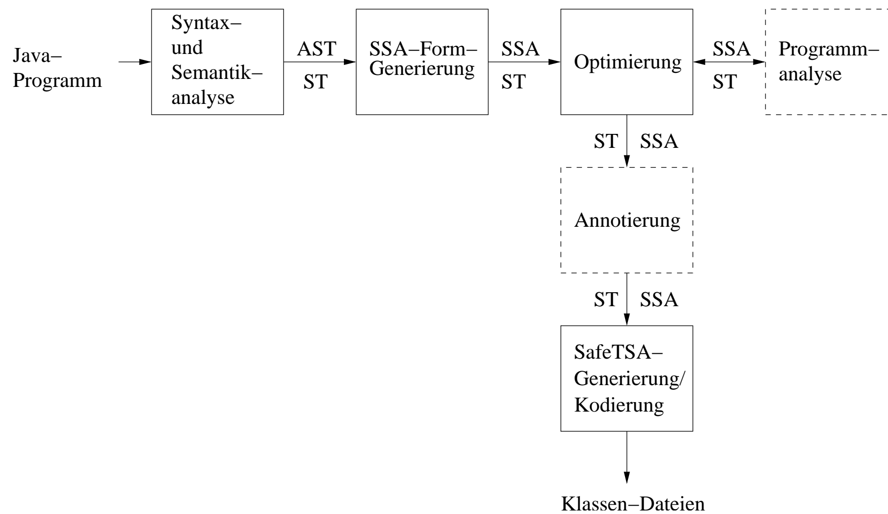


Abbildung 1. Struktureller Aufbau des SafeTSA-Übersetzers

zunächst eine Syntax- und Semantikanalyse aus und transformiert das Eingabeprogramm nach durchgeführter Analyse in einen abstrakten Syntaxbaum (AST) mit zugeordneter Symboltabelle (ST). Nach Transformation in den abstrakten Syntaxbaum wird das zu bearbeitende Programm in SSA-Form gebracht, um nach Durchführung von maschinenunabhängigen Optimierungen in SafeTSA-Format umgewandelt und nach Klassen unterteilt in Dateien abgelegt zu werden. Optional können dem SafeTSA-Format durch Einschalten einer Annotierungskomponente zusätzlich Programminformationen hinzugefügt werden (vergl. etwa [2] oder [3]), die dann vom JIT-Übersetzer zur Durchführung von maschinenabhängigen Optimierungen genutzt werden können.

Bei der Konzeption der Konsumentenseite unseres Systems wurde besonders darauf geachtet, die Systemkomponenten weitestgehend unabhängig von verwendeter Zielarchitektur und Systemumgebung zu realisieren, was eine schnelle Integration der Konsumentenseite in bereits existierende Laufzeitumgebungen unterstützt. Auf der Konsumentenseite wird zunächst vom Dekodierer die SafeTSA-Struktur des Programms wiederhergestellt. Die Überprüfung der Typ- und Referenzsicherheit wird gleichzeitig während der Wiederherstellung des Zwischencodeformats durchgeführt. Der optimierende Codegenerator der Konsumentenseite überführt die verifizierte SafeTSA-Darstellung dann sukzessive in verschiedene Zwischencoderepräsentationen, führt Optimierungen auf diesen aus und bildet in einem letzten Schritt den Maschinencode für die jeweilige Zielarchitektur. Es wurden Konsumentenseiten des Systems für Intels IA32-Architektur und IBMs PowerPC-Architektur geschaffen [4]. Beide Konsumentenseiten wurden in Form eines auf der Zwischencoderepräsentation *SafeTSA* basierenden optimierenden JIT-Compilers für IBMs virtuelle Maschine JikesRVM [5] realisiert.

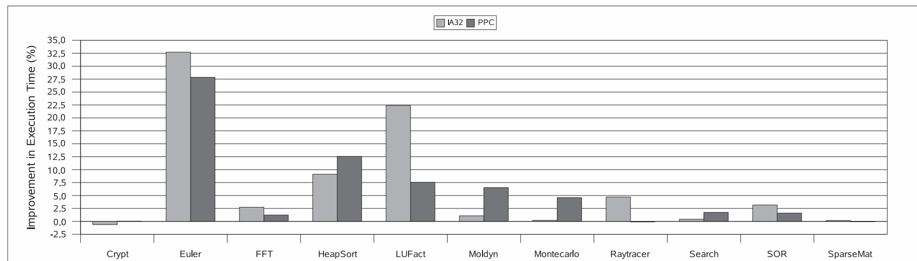


Abbildung 2. Kombination ausgewählter Optimierungen

Zur Auswahl der in die Produzentenseite unseres Systems zu integrieren maschinenunabhängigen Optimierungsarten wurde eine auf den Java-Grande-Forum-Benchmark-Programmen (JGF-Benchmarks) basierende Testumgebung geschaffen. Die JGF-Benchmarks stehen für eine Ansammlung von in der Programmiersprache Java geschriebenen Anwenderprogrammen, die im Jahr 2000 vom *Java Grande Forum* zur Überprüfung der Einsetzbarkeit von Java-Ausführungsumgebungen (JVMs, Java-Übersetzer, Java-Hardware, etc.) für rechenintensive Anwendungen vorgeschlagen wurden [6]. Die zur Überprüfung der Optimierungsarten durchgeführten Messungen wurden auf beiden von der Jikes-RVM unterstützten Rechner-Architekturen durchgeführt. Als Repräsentant der PowerPC-Architektur wurde ein PowerMacG4-Rechner verwendet, der mit einem PowerPC-G4-Prozessor (733 MHz), einem Hauptspeicher von 1,5 GB und einem L2-Cache (256 KB) ausgestattet war. Die Messungen für die IA32-Architektur wurden auf einem Standard-PC ausgeführt, der mit einem Pentium-3-kompatiblen Prozessor (1,5 GHz) und einem Hauptspeicher von ebenfalls 1,5 GB ausgestattet war.

Die auf der PowerPC- und IA32-Architektur durchgeführten Messungen zeigen, dass üblicherweise als maschinenunabhängig eingestufte Optimierungsformen tatsächlich oftmals sehr maschinenabhängiger Natur sind. Tatsächlich konnte keine Optimierungsart gefunden werden, die für beide Architekturen und allen Benchmarkprogrammen aussschließlich zu einer Verbesserung im Laufzeitverhalten führt. Die für die IA32-Architektur erzielten Ergebnisse offenbarten insbesondere, dass die Ausführung von Programmen in starkem Maße von der Anzahl an zur Verfügung stehenden Registern und der für die Registerzuordnung verwendeten Registerallokationsstrategie abhängt. Für SafeTSA-Programme zeigte sich dabei, dass die Durchführung von maschinenunabhängigen Optimierungen für die IA32-Architektur oftmals nur dann Sinn macht, wenn diese nicht über Basisblöcke hinweg durchgeführt werden. Weiter deckten die Messungen auch auf, dass bei den Optimierungen durchgeführte Umstrukturierungen zusätzlich das Cache-Verhalten eines Programms ungünstig verändern und damit dessen Laufzeitverhalten verschlechtern können.

Nichtsdestotrotz belegen die durchgeführten Messungen jedoch auch, dass die Verwendung von maschinenunabhängigen Optimierungen auch im Bereich

des mobilen Codes durchaus einen Nutzen haben kann. Insbesondere zeigen die Ergebnisse, dass die Verwendung von nicht über Basisblöcken hinweg durchgeführten Optimierungen – konkret gesehen, eine Eliminierung von nutzlosen Programmcode, eine Konstantenfortpflanzung, eine Eliminierung von Lade- und Speicherbefehlen sowie eine Eliminierung von gemeinsamen Teilausdrücken – eine sinnvolle Kombination maschinenunabhängiger Optimierungen für den Einsatz im SafeTSA-System sein kann [7].

Abbildung 2 zeigt die mit dieser Kombination an Optimierungsarten erreichbaren Laufzeitverbesserungen relativ zum Laufzeitverhalten von nicht optimierten SafeTSA-Programmen. Wie in der Abbildung zu sehen ist, konnte auf der PowerPC-Architektur für drei Benchmarkprogramme mit den durchgeführten Optimierungen eine Laufzeitverbesserung von mehr als 7,5 % erreicht werden. Auf der IA32-Architektur konnte mit den Optimierungen für vier Benchmarkprogramme das Laufverhalten um über 4 % verbessert werden, wobei für zwei der Benchmarkprogramme sogar eine Laufzeitverbesserung von mehr als 20 % zu beobachten war.

Literatur

1. Amme, W., Dalton, N., Franz, M., von Ronne, J.: SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In: Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation, Snowbird, Utah, USA (June 20–22, 2001)
2. Hartmann, A., Amme, W., von Ronne, J., Franz, M.: Code annotation for safe and efficient dynamic object resolution. In Knoop, J., Zimmermann, W., eds.: Proceedings of Compiler Optimization Meets Compiler Verification (COCV'2003), Warsaw, Poland (April 2003) 18–32
3. Amme, W., Möller, M.A., Adler, P.: Data flow analysis as a general concept for the transport of verifiable program annotations. Electron. Notes Theor. Comput. Sci. **176**(3) (2007) 97–108
4. Amme, W., von Ronne, J., Franz, M.: SSA-Based Mobile Code: Implementation and Empirical Evaluation. ACM Transactions on Architecture and Code Optimization **4**(2) (2007)
5. Alpern, B., Attanasio, C.R., et al.: The Jalapeno virtual machine. IBM System Journal **39**(1) (February 2000)
6. Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S., Davey, R.A.: A benchmark suite for high performance Java. Concurrency: Practice and Experience **12**(6) (May 2000) 375–388
7. Amme, W., von Ronne, J., Adler, P., Franz, M.: The effectiveness of producer-side machine-independent optimizations for mobile code. Softw., Pract. Exper. **39**(10) (2009) 923–946

Programming Support for Cell/BE Multiprocessor

Enes Bajrović and Eduard Mehofer

Institute of Scientific Computing, University of Vienna, Austria,
{bajrovic,mehofer}@par.univie.ac.at
WWW home page: http://www.par.univie.ac.at

Abstract. An emerging class of architectures are accelerator-based heterogeneous multiprocessors with software-managed memory hierarchies like Cell/BE. A major difficulty in programming such kind of machines are the explicit data transfers between the different memories raising new programming challenges. In this paper we discuss a programming approach which supports application programmers in writing efficient code for non-cache-based architectures. A crucial role plays the interplay between the three parties programmer, parallelization framework, and native compiler. Based on our experiences with past programming approaches, we propose language extensions to orchestrate parallel execution of threads and to control data transfers. Experiments are performed to analyze the roles of programmer and compiler in more detail.

1 Introduction

Accelerator-based heterogeneous multiprocessors with software-managed memory hierarchies are getting more and more wide-spread for high performance systems. Usually these hybrid systems consist of standard cores enhanced by dedicated non-general-purpose accelerators with explicitly managed memory hierarchies. Such hybrid architectures raise new programming challenges. Besides distributing the tasks onto the standard CPU and the accelerators (if beneficial), the explicit data transfers between main memory and local memories have to be realized. But even if all the data are in the local memory, efficiency is still a challenging problem, since programming of accelerators is not as simple as programming standard CPUs.

Currently, major research efforts are undertaken by academia and industry to find answers how to deal with these new programming challenges and to leverage the computing power of those multiprocessors. Different approaches are discussed controversially without a consensus within the community.

Even after decades of research, there is still often a large performance gap between automatic parallelization and explicit parallel expert code. In this paper we discuss the interplay between programmer, parallelization framework, and native compiler. We present our system VieCell which assists programming heterogeneous multiprocessors with explicitly managed memory hierarchies, namely the Cell/BE multiprocessor. Our approach reflects the principle that parallelization

must be under programmer control—efficient parallel algorithms can be developed by programmers only and cannot be generated automatically from sequential code. A small number of directives controls parallel execution. In fact, we only add a coordination model to the sequential programming language C. Our application domain are scientific applications which are usually characterized by floating point operations on large data arrays.

The paper is organized as follows. Section 2 discusses the interplay between programmer, parallelization framework, and native compiler which is the key for supporting heterogeneous multiprocessors successfully. The programming approach is illustrated in Section 3 together with an example. Runtime measurements and optimizations for Cell/BE are presented in Section 4. The paper concludes with related work in Section 5, and a summary in Section 6.

2 Efficient and Portable Programming of Architectures with Software-Managed Memory Hierarchies

While chip multiprocessors (CMPs) alleviate problems known as *power wall* or *instruction-level parallelism* (ILP) *wall*, they increase the *programmability wall*. On the one hand, program development for multi-core processors, especially for heterogeneous multi-core processors, is significantly more complex than for single-core processors. On the other hand, programmers have been traditionally trained for the development of sequential programs, and only a small percentage of them have experience with parallel programming. In the past, programmers could trust that compilers succeeded to pass the increased computing power of next processor generations to applications without high porting effort. This was due to relatively homogeneous processor designs even from different hardware vendors with instruction-level parallelism supported at hardware level. The architectural change to CMPs, however, affects the programmer in several ways. On the one hand, thread level parallelism (TLP) must be exploited effectively and efficiently. In general, this cannot be done automatically by a compilation system, but requires assistance by the programmer. On the other hand, multi-core architectures differ significantly requiring that applications must be adapted to the various platforms. This porting problem is worsened by the fact that the average lifetime of hardware is about 5 years, whereas the average lifetime of applications is about 20-30 years.

A crucial role in addressing the programmability wall plays the relationship between the three involved parties programmer, parallelization framework, and native compiler. Experiences in the past have shown the limits of parallelizing compilers. Above all, parallelizing compilers will fail for programs which do not exhibit parallelism, since sequential algorithms have been used. Consequently, parallelization must be under programmer control—efficient parallel algorithms can be developed by programmers only and parallelism shall not be hidden. Directives must be provided for the programmer to control the parallel activities and to manage the explicit data transfers at a high level in a machine-independent way.

Basically a parallel program can be separated into *computation* and *coordination* [8]. The computation model allows a programmer to write a single-threaded computational activity, whereas the coordination model supports thread creation, data transfer, and synchronization. A discussion of integration vs. separation can be found in Gelernter and Carriero [8]. It is highly interesting that most of the arguments apply in these days too. In the year 1992 parallel programming was dominated by message-passing and the need for better programming support. In response to the emerging architectures of that time it is said "diversity with respect to language, hardware platform, physical location ... will be normal in the new era"—a sentence which still applies nowadays.[1]

In the following we summarize the distribution of duties between programmer, parallelization framework, and native compiler.

Programmer. The programmer controls parallelization explicitly. For portability reasons, it is the goal that the code is written in a machine-independent way.

Parallelization framework. The parallelization framework supports the coordination model. The framework realizes tasks like thread management, data transfers, or machine specific optimizations during the parallelization process—tasks which can be handled by an environment successfully and which should not be dealt with by the programmer for portability reasons. Examples of machine specific optimizations are (1) splitting of big data chunks to fit in the small local memories, (2) aggregation of small data transfers to larger pieces, (3) hiding transfers with computation (e.g. double buffering), (4) streaming optimizations, or (5) eliminating synchronization points.

Native compiler. The native compiler optimizes the code assigned to the computing device. In addition to optimizations common for object code compilers, optimizations like vectorization (if a vector unit exists), loop unrolling, or software pipelining shall be supported.

3 Overview of Programming System VieCell

Our system VieCell targets heterogeneous multiprocessors, namely Cell/BE, with a main CPU and a number of accelerators or co-processors with local memories. In case of Cell/BE, the main CPU is the PPU (Power Processor Unit) and the accelerators are called SPUs (Synergistic Processor Units). Data transfers between main memory and local stores are managed explicitly and not implicitly with e.g. load/store instructions. Typically, the main processor and the accelerators have different instruction sets.

Parallelization is fully controlled by the programmer. Since the computation model is covered by programming language C, the coordination model has to be addressed only. The extensions for the coordination model have been realized

[1] By the way, message passing is still the dominating programming paradigm for scientific applications.

with directives embedded in the sequential language. Ignoring the directives results in a semantically equivalent sequential version of the program.

The basic computational unit which can be executed in parallel is an SPU function extended by a parameter in/out-description which are spawned on the SPUs. The data transfers take place at function invocation and return, and constitute the execution context of the SPU function. Hence, data transfers are aggregated to larger pieces which reflects the shopping-list parallelization strategy as proposed by Cell/BE chief scientists [10] for such kind of architectures.

For Cell-like architectures it is an obvious approach that at program start-up a single master thread is created on the PPU which exists for the duration of the whole program and which starts executing the program sequentially. When the master thread encounters a `parallel` loop, slave threads are created for each SPU to control parallel execution. The task of each slave thread is to load the executable of an SPU function onto the SPU, transfer at the beginning the data to the local memory, and write the result back to main memory. After termination of all slave threads, the master thread in the PPU continues execution. Thus the PPU acts as orchestrator responsible for realizing work distribution and coordinating parallel execution.
Parallel execution is controlled by a small number of directives:

- **pragma parallel**. When the master thread reaches a `parallel` loop, the PPU loads the binary of the SPU function onto the SPUs and distributes the work between the SPUs. The body of a `parallel` loop contains exactly one SPU function call and the programmer asserts that it is legal to execute the function in parallel.
- **pragma public**. An SPU compilation unit contains exactly one function with the *public*-attribute, called *SPU function*, which is invokable from the PPU.
  The *parameter clause* specifies for each parameter whether it is an *in*, *out*, or *inout* parameter together with the number of data elements to be transferred. The semantics of the parameter transfer is call-by-value-result, i.e. the arrays are copied between main memory and local memory forward and backward.
- **pragma comm**. The *communication*-attribute indicates that data structures allocated by the PPU (PPU compilation unit) will be transferred between PPU and SPU. This attribute is used to take care of alignment.

Since we are targeting stream-like applications in the field of computational science, we provide additional notations to steer optimizations for such kind of applications. As an example a parallel matrix-vector multiplication is shown in Fig. 1.

4 Experiments on Cell/BE

In this section we discuss experimental results performed on single SPU of a Cell/BE using native IBM XL C/C++ XLC compiler version 10.1 and GNU

```
01  #pragma vie comm
02  float A[M][N], X[N], Y[M];
    :
    :  sequential execution
03  #pragma vie parallel
04  for (int i=0; i<M; i++)
05    SPU_dot_pr(&A[i][0],X,&Y[i]);
    :
    :  sequential execution
```

(a) PPU user code.

```
01  #pragma vie public vec1(in,N), \
02                     vec2(in,N), \
03                     vec3(out,1)
04  void SPU_dot_pr(float vec1[],
05                  float vec2[],
06                  float vec3[])
07  {
08    float sum=0;
09    for (int j=0; j<N; j++) {
10      sum+=vec1[j]*vec2[j];
11    }
12   vec3[0]=sum;
13  }
```

(b) SPU user code.

Fig. 1. Matrix-vector multiplication.

GCC open source compiler shipped with Cell SDK 3.1 (both with `-O3` optimization flag).

Cell/BE is a heterogeneous multiprocessor with an IBM Power processor core, called PPU (Power Processor Unit), and 8 specialized accelerators or co-processors with local stores, called SPU (Synergistic Processor Unit). The PPU and SPU have different instruction sets and the SPU contains a SIMD execution unit. The small local store of 256 KB holds instructions and data and is the only memory directly addressable by the SPU.

The experiments have been conducted on an IBM BladeCenter QS22 with two IBM PowerXCell 8i processors (3.2GHz/1MB L2) mounted in an IBM BladeCenter H chassis. IBM PowerXCell 8i processor is the follow-up model of the Cell processor with much better double-precision floating point performance.

As example we take vector-vector addition and start with a straight-forward implementation (var. A) as shown in Fig. 2(a) and compile it with XLC and GCC. With XLC we got about 3.72 GFlop/s, whereas for GCC we got only 0.13 GFlop/s. The results of XLC are significantly better than for GCC, but the 3.72 GFlop/s obtained by XLC seem to be low either compared to the peak performance of 25.6 GFlop/s.[2]

For vector-vector addition, however, FMA operations cannot be used resulting in 50% of peak performance. Further, since there are three load/store operations vs. one floating-point operation, only one third can be achieved resulting in a maximum sustained performance of 4.27 GFlop/s.

[2] For single precision floating point 2 operations (FMA) are performed on each of the floats in the quad-word per cycle, leading to 3.2GHz * 8 = 25.6 GFlop/s.

```
01 float a[n];
02 float b[n];
03 float c[n];
04
05 for (i = 0; i < n; i++)
06     c[i] = a[i] + b[i];
```

(a) Scalar code (var. A).

```
01 vector float a[n];
02 vector float b[n];
03 vector float c[n];
04
05 for (i = 0; i < n/4; i++)
06   c[i] = spu_add(a[i], b[i]);
```

(b) Vectorized code (var. B).

```
vector float x0,x1,x2,x3,x4,x5;
vector float y0,y1,y2,y3,y4,y5;
vector float z0,z1,z2,z3,z4,z5;
    ⋮  pre-loop code
for (i = 0; i < n/4 - 2; i+=6) {
    // Store [i] - [i+5]
    c[i+0] = z0; c[i+1] = z1; c[i+2]=z2;
    c[i+3] = z3; c[i+4] = z4; c[i+5]=z5;
     // Compute [i+1]  [i+6]
    z0=spu_add(x0, y0); z1=spu_add(x1, y1); z2=spu_add(x2, y2);
    z3=spu_add(x3, y3); z2=spu_add(x2, y2); z3=spu_add(x3, y3);
     // Load next a: [i+12] to [i+17]
    x0 = a[i+12]; x1 = a[i+13]; x2 = a[i+14];
    x3 = a[i+15]; x4 = a[i+16]; x5 = a[i+17];
    // Load next b: [i+12] to [i+17]
    y0 = b[i+12]; y1 = b[i+13]; y2 = b[i+14];
    y3 = b[i+15]; y4 = b[i+16]; y5 = b[i+17];
}
    ⋮  post-loop code
```

(c) Software pipelining with unrolling (var. E).

Fig. 2. Vector-vector addition.

Now we try to hand-optimize the code to get with GCC similar performance results like XLC. First we vectorized the code (var. B) as shown in Fig. 2(b). For XLC nothing changed, while with GCC we got 0.78 which is a speedup by a factor of 6, but still far away from XLC.

Explicit loads and stores to enable software pipelining (var. C) resulted in doubling the performance of the code compiled with GCC, with 1.58 GFlop/s, which is a speedup of 12.15 compared to the initial code variant A.

Next, we apply software pipelining and a technique similar to loop unrolling by a factor 2 (var. D) and factor 6 (var. E) as shown in Fig. 2(c). The SPU has two distinct instruction pipelines supporting dual-issue. Load/store instructions move the data from local store to registers and back with the latency of 6 cycles. On the other hand, floating-point operations take exactly the same amount of cycles. If we consider two pipelines and only loads/stores and floating-add operations, the instruction flow looks like in Fig. 3. With this optimization we got for GCC 3.72 GFlop/s which is a speedup of 30, and for XLC approximately the same result.

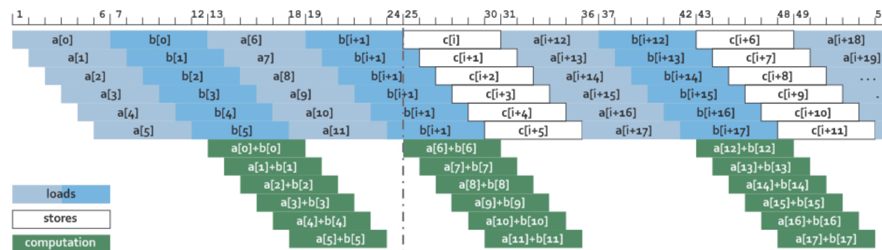


Fig. 3. Software pipelining with unrolling.

Finally, we succeeded for GCC to obtain similar performance numbers like for XLC, however, the hand-optimized code is much more complex compared to the initial code version. A summary of the performance numbers can be found in Fig. 4 and a graphical presentation is shown in Fig. 5.

The experiments have shown that XLC manages the complexity of the low-level optimizations successfully resulting in good performance numbers. Obviously, optimizations performed by the programmer in order to compensate compiler deficiencies results in non-portable complex code. Further, programmer productivity decreases as well.

5 Related Work

Many research groups from the parallel computing community as well as graphics community work on programming of accelerator-based heterogeneous multiprocessors. Related approaches include CUDA from Nvidia [13], Brook for GPUs

Program variants	GCC	XLC	Speedup GCC/XLC
Scalar (A)	0.13	3.72	1.00/1.00
Vectorized (B)	0.78	3.72	6.00/1.00
SW Pipelining (C)	1.58	3.72	12.15/1.00
SW pipelining with prefetching 2 elemets (D)	1.71	3.81	13.15/1.00
SW pipelining with prefetching 6 elemets (E)	3.72	3.84	30.00/1.03

Fig. 4. Performance results as table.

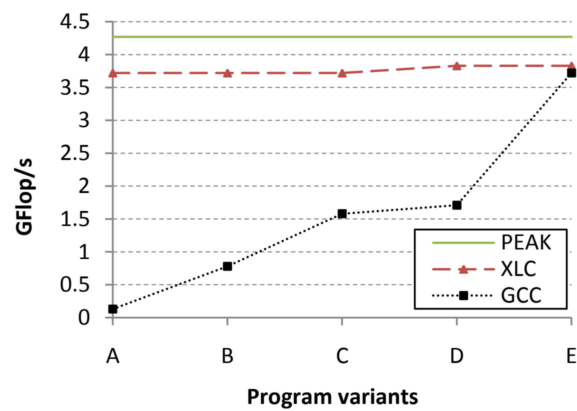


Fig. 5. Performance results: graphical presentation.

from Stanford University [2], StreamIt from MIT [14], HMPP from CAPS entreprise [6], and RapidMind platform from the very same company [11]. Other well-known parallel programming languages include UPC (Unified Parallel C) [5], CAF (Co-array Fortran) [12], Titanium (Java-based) [9], and Sequoia [7]. Higher level of abstractions provide the languages of the HPCS (High Productivity Computing Systems) program of DARPA: X10 (IBM) [4], Chapel (Cray) [3], and Fortress (Sun) [1].

6 Conclusion

We discussed the importance of the interplay between the three parties programmer, parallelization framework, and native compiler which is the key for supporting heterogeneous multiprocessors successfully. The programming approach presented in this paper addresses Cell/BE like architectures and is based on a coordination model added to C. The separation between coordination and computation fits specifically to architectures like Cell/BE with PPU as main unit orchestrating the parallel activities and SPUs as accelerators; or even in bigger contexts like the Los Alamos Roadrunner architecture[3] with Opterons as main units and orchestrators and Cell/BE multiprocessors as accelerators.

References

1. Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 1.0 edition, March 2008.
2. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.
3. B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.
4. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.
5. UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.
6. Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007.
7. Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[3] Since Nov 2008 the fastest supercomputer in the TOP500 list and the first computer breaking the petaflop/s performance barrier.

8. David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.
9. Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, EECS Department, University of California, Berkeley, Nov 2005.
10. Peter Hofstee – *An Interview.* Custom Processing. *ACM Queue*, 5(1), 2007.
11. Michael D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx Multi-core Applications Conference*, Santa Clara, CA, October-November 2006.
12. Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.
13. NVIDIA CUDA. NVIDIA, http://developer.nvidia.com/ object/cuda.html.
14. William Thies. *Language and Compiler Support for Stream Programs*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2009.

SATIrE within ALL-TIMES: Improving Timing Technology with Source Code Analysis

Gergö Barany

Institute of Computer Languages, Vienna University of Technology

Abstract. We present the SATIrE source-to-source analysis framework within the context of ALL-TIMES, a European research and development project aimed at improving and integrating existing tools in the area of timing analysis. Within the project, SATIrE contributes by performing source-level static analysis on C programs and exporting its results for other tools to use.
This work gives an overview of SATIrE and how its analyses may improve timing analysis results obtained by other tools. We discuss SATIrE's efficient and powerful context-sensitive unification-based points-to analysis and its value interval analysis. We explain how SATIrE's integration with other analysis tools handles issues such as combination of source-level with binary-level analysis and communication of views of function call contexts between tools.

1 Introduction

With safety-critical embedded real-time systems controlling automobiles and airplanes, timing errors can have disastrous consequences. It is therefore very important to ensure that any hard deadlines specified for parts of the system can be met under all possible circumstances. Timing analysis aims to predict worst-case timing behavior in order to give feedback to developers and provide information to validation processes.
There are many forms of timing analysis, each with a number of advantages and disadvantages. Static analysis of source code benefits from being aware of symbolic variable names, data types, and program structure. However, actual *timing* results cannot be derived from the source code alone, since it does not uniquely determine the actual machine instructions that will be executed on the target platform. Hence, static analysis of the binary is needed to derive worst-case timings for code snippets. This level of analysis can make use of knowledge about actual machine code in conjunction with machine parameters such as instruction timings and predicted worst-case cache and pipelining behavior.
In another dimension, dynamic analysis relies on running the system to collect *observed* timings and other system characteristics, given certain inputs which are expected to cover worst-case circumstances. Dynamic analysis often results in tighter timings than static analysis, but it typically lacks a guarantee that the tested cases covered the worst-case behavior.

The goal of the ALL-TIMES project is to integrate various European tools which cover different points in this space of possible timing analysis approaches. This paper describes a part of the overall effort in ALL-TIMES, focusing on the SATIrE system and its connections to other tools within the project. The following section describes the SATIrE framework, the analyses it provides, and ways to extend it with further analyses; Section 3 discusses how SATIrE supports other tools in the ALL-TIMES project by exporting analysis information. Section 6 concludes.

2 The SATIrE System

SATIrE (Static Analysis Tool Integration Engine)[1] is a framework for integrating various static analysis and source code manipulation tools. Its focus is on programs written in the C programming language. SATIrE has been under development at Vienna University of Technology since late 2004.

2.1 Framework

SATIrE allows users to build tools that analyze or transform C programs. To this end, it provides interfaces to the ROSE[2] source-to-source transformation system; ROSE includes the C and C++ frontend by Edison Design Group and provides an object-oriented abstract syntax tree (AST) for manipulation. As an alternative frontend, SATIrE recently acquired bindings to clang[3]; clang's output can be translated by SATIrE into a ROSE AST.
Given the AST representing a program, there are several ways of analyzing and manipulating it. ROSE includes a number of analyses and transformations, including a loop optimizer that can perform common operations such as loop unrolling. To allow data-flow analysis on C programs, SATIrE includes a component that builds an interprocedural control-flow graph (ICFG) and bindings to the Program Analyzer Generator (PAG)[4] which generates data-flow analyzers from functional specifications. SATIrE also includes tools to export ASTs as Prolog terms and vice versa, to enable program analysis and transformation using the Prolog programming language.
ASTs that have been transformed or annotated with analysis results (in the form of comments or `#pragma` statements) can then be unparsed to C source code for further processing with other tools or compilers. The overall architecture of SATIrE is shown in Figure 1.

[1] `http://www.complang.tuwien.ac.at/satire/`
[2] `http://www.rosecompiler.org`
[3] `http://clang.llvm.org`
[4] `http://www.absint.de/pag/`

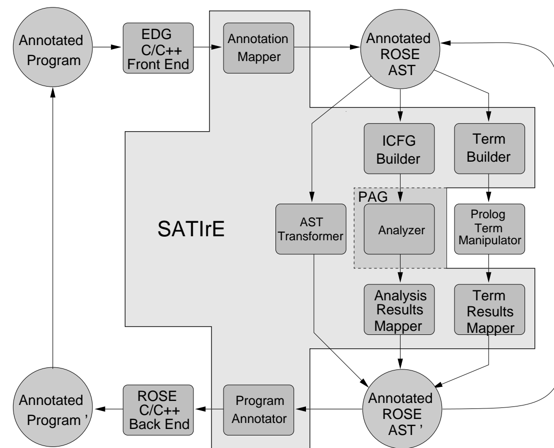


Fig. 1. Architecture of the SATIrE program analysis framework. The central program representation is the possibly annotated AST provided by ROSE. Various toolchains can work on this representation, possibly with intermediate transformations. The final result can be unparsed to annotated source code.

2.2 SATIrE Analyses

Besides providing interfaces for user-defined analyzers, SATIrE also includes a number of predefined analysis stages. The most important ones are the points-to analyzer, the value interval analyzer, and the loop bounds analyzer.

Points-To Analysis SATIrE's points-to analysis is a flow-insensitive unification-based analysis inspired by Steensgaard [Ste96]. The basic analysis performs a single pass over the program, assigning an abstract 'location' to each program variable, function, or dynamic memory allocation site. The effects of pointer assignments are modeled using points-to edges between locations. Each location is constrained to have at most one outgoing points-to edge; if some location might point to two or more different locations, those locations are merged into a new combined location.
This merging ensures that the analysis can be implemented in almost-linear time using a fast Union/Find data structure [Tar75]. However, it is also a source of imprecision as it may introduce spurious points-to relations that cannot be realized in any actual run of the program.
The basic algorithm suffers from the fact that it is context-insensitive; if a function that receives a pointer argument is called at several different sites, the analysis will merge all the objects that may be pointed to at any call site. Since passing and returning pointers is very common usage

in C, this can lead to considerable imprecision. Much of the lost precision can be regained by making the analysis context-sensitive: Based on some notion of interprocedural context, each function is analyzed multiple times. Argument and return-value locations of the function contexts are linked according to the calling relations between the contexts. This cloning approach is similar Lattner et al.'s context-sensitive points-to analysis [LLA07]. Currently, SATIrE uses context information derived by PAG, which can compute call strings of any bounded or (for non-recursive programs) arbitrary length.
Any external functions called from the program must have summaries available to the points-to analysis. Otherwise, they are treated conservatively as calling an unknown function that may introduce arbitrary aliases to global variables. The current implementation provides fully polymorphic summaries for part of the C standard library. Summaries are currently built into the analyzer, but an external annotation language for functions is under consideration.

Value Interval Analysis The interval analysis (sometimes called 'value range analysis') integrated in SATIrE derives possible values of integer variables in C programs. Each variable is associated with a pair of numbers representing an interval of values, which may be open on either side (lower bound of $-\infty$ or upper bound of ∞). The analysis is flow-sensitive, meaning that analysis information specific to every program point is derived. If at some point a variable is associated with an interval with bounds $[a, b]$, this means that at that point, the variable's value is definitely somewhere between a and b. This analysis is conservative, as in many cases not all values between a and b can actually be realized in executions of the program. However, all possible actual values are covered by the intervals derived by the analysis. The interval analysis is implemented as an abstract interpretation [CC77]. The declarative analysis specification is translated to an executable program using PAG.
Constant values in interval bounds typically come from direct assignment of a constant to a variable, such as initializing a loop counter to 0, or from comparison of a variable with a constant. For instance, a loop condition `i <= 10` gives an upper bound for the variable on the path entering the loop, and a lower bound for the path leaving the loop. In certain cases, the analysis can also make use of `assert` statements in the program that were inserted by programmers with domain knowledge, or by some other program analysis/transformation. The information in a statement like `assert(x >= 0 && x <= 10);` can be used by the interval analysis to infer that at the program point following the statement, the value of variable `x` must be in the range $[0, 10]$, regardless of what was known about its value before.
The analysis is integrated with SATIrE's points-to analysis. Integer assignments or reads through pointers can therefore be resolved to sets of possibly referenced variables. This can lead to much more precise results than other approaches, which cannot handle pointers and must make very conservative assumptions. In SATIrE's interval analysis, if only one variable can be referenced, its associated interval can be used or updated

as for direct reads or assignments; otherwise, the intervals for all concerned variables are read or updated in a conservative way that never underestimates the actual possible ranges of values. Similarly, an array variable is treated like a set of variables, and assignments to array elements do not replace but rather extend the interval representing all possible values stored in the array ('weak update').
The interval analysis is inter-procedural, i. e., intervals associated with argument expressions of function calls are propagated into the corresponding functions. Using facilities provided by PAG, the interval analysis can be used in a context-sensitive way with arbitrarily long call strings.

Loop Bounds Analysis SATIrE includes a component which computes loop bounds for loops based on iteration variables [PKST08]. It uses results of the interval analysis and structural information about the program to build a set of inequalities, which are solved to yield constraints on the numbers of loop iterations.
The loop analyzer constructs a set of inequalities for certain loops. Essentially, it looks for loops preceded by the initialization of an iteration variable, an exit condition consisting of an inequality involving the variable (or a set of such inequalities connected by 'logical or' operators), and exactly one increment or decrement of the variable inside the loop with a bounded step size. Experience shows that most loops in embedded systems software are of this form, so the analysis is widely applicable in the domain covered by the ALL-TIMES project.
Assuming the loop variable is i, the initialization expression is Init, the test expression is $i \leq \mathit{Test}$, and the step size Step is known to be positive, the following equalities and inequalities are generated:

$$i \geq \mathit{Init} \qquad i \leq \mathit{Test} \qquad (i - \mathit{Init}) \bmod \mathit{Step} = 0$$

The number of distinct solutions of this system is just the maximum number of possible iterations of the loop. An external constraint solver is used to calculate the number of solutions, which is often considerably more efficient than simply enumerating the solutions. Currently, SATIrE uses the *clpfd* solver distributed with SWI-Prolog[5] to calculate these loop bounds.
Applying the basic analysis recursively from outer to inner loops, nested counting loops can be analyzed as well. The loop bound for each inner loop is given in terms of the scope containing the outer loop. In particular, this means that triangular loops are not overestimated.

3 Integration in ALL-TIMES

This section describes the ALL-TIMES project and details SATIrE's involvement in the project.

[5] `http://www.swi-prolog.org`

3.1 ALL-TIMES Project Partners

ALL-TIMES comprises a total of six partners, four commercial companies and two university groups. Each partner contributes an existing tool with a specific approach to timing analysis and corresponding strengths and weaknesses. The partners have identified a number of valuable integrations between tools to combine and exchange various kinds of analysis data in order to obtain better results.
The partners and their tools are:

AbsInt Angewandte Informatik GmbH [6] develops the **aiT** family of WCET analyzer tools. aiT performs static analysis directly on the actual application binary. Using abstract interpretation, aiT derives possible value ranges of registers and memory locations; it also computes upper bounds on WCET of basic blocks, taking cache and pipelining effects into account. Using integer linear programming (ILP), aiT then derives a worst-case program path and the corresponding WCET bound from the basic block estimates.

Gliwa GmbH [7] is the developer of **debugGURU**, a dynamic analysis framework. It uses instrumentation of the target application to collect timing information at run-time. Using plugins, debugGURU can collect various kinds of information, including execution times and memory usage. The collected information can be communicated to host PCs using industry-standard bus systems.

Symtavision GmbH [8] provides **SymTA/S**, a system-level timing and scheduling analysis tool. SymTA/S works with a high-level abstract model of the application in terms of program tasks and resources such as CPUs and buses. It calculates resource loads, worst-case response times for tasks and worst-case transmission times for messages.

Rapita Systems Ltd [9] produces the **RapiTime** toolkit for dynamic analysis. RapiTime instruments the application with measurement code and uses the measurement to profile performance, provide code coverage information, and perform WCET analysis.

Mälardalen University's WCET group[10] uses its **SWEET** tool for timing analysis. Its flow analysis is based on abstract execution and can derive complex flow constraints relating execution frequencies for different points in the program. SWEET analyzes programs in the ALF representation [GEL+09], which is designed to be generated either from source code or from a binary.

Vienna University of Technology, compilers and languages group[11], contributes **SATIrE**, the framework for source-based analysis and transformation of C programs described in Section 2.

[6] `http://www.absint.com`
[7] `http://www.gliwa.com`
[8] `http://www.symtavision.com`
[9] `http://www.rapitasystems.com`
[10] `http://www.mrtc.mdh.se/projects/wcet/`
[11] `http://www.complang.tuwien.ac.at`

4 SATIrE's Connections in ALL-TIMES

This section explains the tool integrations in the ALL-TIMES project that SATIrE is involved with.

4.1 Integration: SATIrE-RapiTime

The integration of SATIrE with RapiTime involves communication of analysis information from SATIrE to RapiTime. The main goal is to provide information about function pointers, while information regarding loop bounds may also be useful to RapiTime.

As RapiTime uses dynamic analysis to gather information at run-time, one cannot always be sure that all possible executions of certain parts of the code have been covered by its analysis. RapiTime therefore provides the possibility for users to annotate the program's source code with high-level knowledge about issues such as points-to relations or flow constraints. Within the ALL-TIMES project, SATIrE will use static analysis to compute some of the information that would otherwise be provided manually. This saves users from part of the tedious and error-prone task of program annotation.

In order to compute a worst-case timing for a function call, RapiTime must know all the possible functions that may be called at that site (in a certain context). Since embedded systems codes often contain indirect calls through function pointers, this information is typically not immediately available. During execution of the system, the code annotated by RapiTime can record all *observed* functions called from a certain site, but as noted above, it may not always be sure that these were all the *possible* call targets for that call site. Without this information, it must make a conservative approximation or reject the program.

This is where analysis information from SATIrE can aid RapiTime in deriving tight WCETs: Its points-to analysis can give static information about possible call targets at each call site. The automatic analysis is much faster and more reliable than manual annotations; this is particularly true for context-sensitive annotations. Thus SATIrE's information can tell RapiTime whether it has observed all possible call targets during its tests, or which other possible targets to take into account for its WCET calculation.

Similarly, RapiTime may observe certain numbers of iterations for loops in the application. SATIrE's static analysis of loop bounds may confirm that the observed iterations are indeed the worst case, or provide information for appropriate computation of a guaranteed time bound.

While RapiTime allows users to annotate source code with `#pragma` statements containing analysis information, SATIrE's analysis information will be communicated using an external file format. Source code annotations are very natural for manual annotations, but for automated exchange of information an external format separated from the source code may offer more flexibility.

4.2 Integration: SATIrE-aiT

For the integration with aiT, SATIrE will provide analysis information on points-to relations, unreachable code, and its view of interprocedural contexts, which differs from aiT's.
The static analysis performed by aiT on the application binary takes a purely numeric view of data, including pointers: Pointers are treated as numeric addresses, and aiT's value analysis abstracts their possible values as numeric intervals. This is especially problematic for global entities such as global variables or functions, where a symbolic analysis based on names can be much more exact that pure numeric analysis.
Consider, for instance, a function pointer which may point to two functions `f` and `g`. If the compiler's code layout is such that these functions are not adjacent in the binary, a numeric analysis will determine that the pointer may point to the address of `f` or the address of `g` or any other function that lies between them in the code. Those functions in-between, however, are not actually feasible targets but rather just artifacts of a numeric analysis. Since SATIrE's points-to analysis works on names rather than addresses (which do not even exist on the source level), it is not susceptible to this kind of spurious result. The same reasoning applies to the treatment of pointers to data, which is why SATIrE may also derive much more precise analysis information for pointers to global buffers or stack variables.
As noted in Section 2.2, SATIrE's interval analysis is integrated with its pointer analysis. Thus, in certain cases, the more precise pointer information in SATIrE can be used to derive variable values that aiT is not able to compute. Such values can then be used to identify branch conditions that are always true or always false in certain contexts, and this more precise flow information can result in a better WCET estimation.
Other kinds of information SATIrE exports to aiT are points-to relations for data pointers and possible targets of calls through function pointers.
An important issue in integrating SATIrE and aiT is the fact that these tools have different notions of interprocedural contexts. In aiT's view of the program, loops are transformed into special tail-recursive procedures; a jump back to the loop's head for another iteration corresponds to a tail call. This approach allows aiT to use call strings to model loop contexts, which means that it need not merge analysis information from the first N loop iterations with the information from later iterations. On the other hand, this handling of loops as calls means that direct exchange of call strings between the two tools is not possible.
To communicate context information to aiT, SATIrE uses aiT's concept of 'user-defined registers'. These are pseudo-registers that have no counterpart in the actual code and hardware, but rather only contain values at analysis time. Annotations to the code may refer to, and store, values in these user registers. aiT's value analysis propagates these values just like it propagates values of actual hardware registers and memory locations.
Context information from SATIrE is then communicated as follows: Each function in the program has an associated 'call site' user register. Each call site is associated with a unique numeric identifier. At each call site,

annotations from SATIrE instruct aiT to store the call site register for each possible target function with that site's ID. Within called functions, annotations may be qualified to hold only if the call site registers contain certain specific values. For instance, an annotation restricted to be true only if three registers contain certain call site identifiers essentially corresponds to an annotation that is specified to hold only given a certain three-element call string. The analogy to finite call strings only breaks down in the presence of recursion, in which case having just one register per function makes the analysis less precise—however, recursion is not a major concern in typical embedded control applications.
Information from SATIrE is communicated to aiT using source code annotations, or external annotations that refer to source code locations. aiT does not itself work with source code; rather, it relies on the compiler that generates the binary to also output debug information that relates source code positions to addresses in the executable.

4.3 Integration: SATIrE-SWEET

The integration of SWEET and SATIrE involves various issues. First, as SWEET works on programs in the ALF representation [GEL$^+$09], it needs translators to ALF in order to be able to analyze binaries or source code. One part of the connection between SATIrE and SWEET is therefore a C-to-ALF compiler called melmac[12]. This is a fairly regular C compiler backend, but its tight integration with SATIrE also allows it to output analysis results and meta-information that is useful for the other aspects of the connection.
Thus for the second part of the connection, melmac also outputs information mapping ALF code positions (identified by jump labels) to SATIrE's internal position identifiers as well as source code locations, and information on the call strings used by SATIrE. Using this information, SATIrE's context-sensitive analysis results regarding points-to information and value intervals can also be communicated to SWEET. In contrast to the other connections described above, SWEET and SATIrE have similar notions of program objects (ALF allows named, scoped variables like C does). Thus SATIrE's analysis information referring to program variables and pointer relations is directly useful for SWEET. The tight correspondences between program positions as well as variables allow SATIrE to exchange flow-sensitive interval information with SWEET, which is not the case for the other connections. This can ease the implementation burden on SWEET's developers, who at the time of writing do not have a context-sensitive points-to analysis.
The third and final part of this connection involves communication of analysis information from SWEET to SATIrE. SWEET's strength lies in deriving complex flow facts using abstract execution; these facts identify mutually exclusive program points or give constraints on the relative number of executions of different program points. Currently, SATIrE itself cannot make use of this kind of detailed flow information; however, it can output the flow facts for use by aiT. Thus SATIrE can play the

[12] `http://www.complang.tuwien.ac.at/gergo/melmac/`

role of a translator in a connection involving two other tools that might otherwise not be able to exchange information.

5 Related Work

Gustafsson et al. [GLS+08] give a more detailed overview of the ALL-TIMES project and the tools involved. Schordan [Sch08] presents the SATIrE system in more detail and considers some challenges of annotating source code with analysis information.
There does not appear to be much previous work that deals specifically with integration of source code analysis and WCET calculation. The TuBound tool [PSK08] is an exception: TuBound integrates a source-based tool which derives loop bounds and adds corresponding source code annotations, and a C compiler which includes a component for WCET computation that can make use of these annotations. The source-based part of TuBound is built on SATIrE and the same analyses that SATIrE contributes to ALL-TIMES; however, TuBound's integration of a WCET-aware compiler is different from any of the work reported here.

6 Conclusions

We have presented the SATIrE program analysis framework and described its role within ALL-TIMES, a project aimed at integrating European timing analysis tools. In this project, SATIrE exchanges information with tools that employ a wide range of static and dynamic analysis techniques involving different levels of source, binary and intermediate code. Each connection allows SATIrE to aid the other tools in specific ways by providing valuable information derived using static analysis of source code. The benefits of this integrated approach are expected to validate the basic premise of the ALL-TIMES project, which is that combination of different approaches can yield much better results than each approach in isolation.

References

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[GEL+09] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – a language for WCET flow analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, June 2009.

[GLS+08] Jan Gustafsson, Björn Lisper, Markus Schordan, Christian Ferdinand, Marek Jersak, and Guillem Bernat. ALL-TIMES - a European project on integrating timing technology. In *Proc. Third International Symposium on Leveraging Applications of Formal Methods (ISOLA'08)*, pages 445–459. Springer, October 2008.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 278–289, New York, NY, USA, 2007. ACM.

[PKST08] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level WCET analysis. In *The 18th Workshop on Logic-based methods in Programming Environments (WLPE 2008)*, pages 77–89, Udine, Italy, December 2008.

[PSK08] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 141–148, Prague, Czech Republic, 2008. Österreichische Computer Gesellschaft.

[Sch08] Markus Schordan. Source-to-source analysis with SATIrE - an example revisited. In *Scalable Program Analysis*, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

Acknowledgements

This work is supported by the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No. 215068 funded by the 7th EU R&D Framework Programme. See the project web site at `http://www.all-times.org` for more information about ALL-TIMES.

Computing and Visualizing Closure Objects using Relation Algebra and RelView

Rudolf Berghammer and Bernd Braßel

Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel
{rub|bbr}@informatik.uni-kiel.de

Abstract. Closure objects play an important role in mathematics and computer science. We develop relation-algebraic specifications to recognize several classes of them, compute the complete lattices they constitute and transform any of these closure objects into another. All specifications are algorithmic and can directly be translated into the language of RelView. We show that the system is well suited for computing and visualizing closure objects and their lattices.

1 Introduction

The procedure of computing the closure of a given object is an important basic technique for applications in mathematics and computer science alike. The approach followed most frequently is to employ so-called closure operations. Examples include operations that yield the transitive closure of a relation or the subgroup generated by a set of elements of a given group. The practical importance of closures is also documented by the fact that equivalent or at least very similar notions are frequently reinvented. For example, what [6] denotes as "full implicational system" is called a "full family of functional dependencies" in the theory of relational databases [7] or a "closed family of implications" in formal concept analysis [9]. Likewise a "dependency relation" [6, Section 2.2] is the same as a "contact relation" in the sense of [1] and one needs only to transpose such a relation and restrict its range to $2^X \setminus \{\emptyset\}$ in order to obtain an "entailment relation" in the sense of [8]. Moreover, on the lattice-theoretic side it has been proven that there exists a close correspondence between all of the aforementioned concepts and the concept of a closure operation on a set. All of these objects form complete lattices which are pairwise isomorphic [6].

The subject of this paper (a short version of [3], where all proofs can be found) is to give a relation-algebraic representation of closure objects. The resulting specifications are algorithmic and directly lead to corresponding programs for RelView [5], which is a computer algebra system for the special purpose of computing with relations. The developed formulas provide three basic algorithms for each closure object given as a finite relation, viz. *recognition of an object*, i.e., deciding whether a given relation is, for example, a dependency

relation, *transformation between closure objects*, e.g., compute the closure operation corresponding to a given full implicational system, and *computation of the complete lattice* which is constituted by the set of all such closure objects. Employing the resulting programs, the computer algebra system RelView supports the study of specific closure objects in several ways, among which are *visualization* as Boolean matrices or as graphs, providing various graph layout algorithms and offering a number of ways to highlight selected portions, *retracing* the results of sub-expressions via step-wise execution, and *testing*, e.g., by providing random matrices of a specified degree of filling. Beyond the possibility to study individual closure objects, the presented algorithms scale well and are applicable to large examples. This is due to the fact that finite relations can be implemented efficiently with BDDs [4] and we have taken care that the developed formulas fit the setting of RelView. The only exception is the computation of all possible closure systems which is presented in Section 3.1, as the according problem is well known to be exponential for sets.

2 Relation Algebra

Since many years relation algebra [11, 10] is used by many scientists as conceptual and methodological base of their work. Its importance is due to the fact that many objects of discrete mathematics can be seen as specific relations. Relation algebra allows concise and exact problem specifications and extremely formal and precise calculations that drastically reduce the danger of making mistakes. In this section, we recall the basics of relation algebra, provide the reader with an introduction of how pairs and products can be modeled and show how to represent sets. More specific to the task at hand, its last part deals with the relation-algebraic specification of extremal elements of ordered sets and lattices.

2.1 Relations and Relation Algebra

We write $R : X \leftrightarrow Y$ if R is a relation with domain X and range Y, i.e., a subset of $X \times Y$. If the sets X and Y of R's *type* $[X \leftrightarrow Y]$ are finite, we may consider R as a Boolean matrix. Since this interpretation is well suited for many purposes and is also used by the computer algebra system RelView as one of its possibilities to depict relations, we often use matrix terminology and matrix notation. Especially we speak about the rows, columns and entries of a relation and write $R_{x,y}$ instead of $\langle x, y\rangle \in R$ or $x\,R\,y$. Relation algebra knows three basic relations. The *identity relation* $\mathsf{I} : X \leftrightarrow X$ satisfying for all $x, y \in X$ that $\mathsf{I}_{x,y}$ iff $x = y$, the *universal relation* $\mathsf{L} : X \leftrightarrow Y$ holding for all $x \in X$ and $y \in Y$, and the *empty relation* $\mathsf{O} : X \leftrightarrow Y$ which holds for no pair in $X \times Y$. The *transposition* of a given relation $R : X \leftrightarrow Y$ is denoted by $R^\mathsf{T} : Y \leftrightarrow X$ and satisfies for all x, y that $R^\mathsf{T}_{x,y}$ iff $R_{y,x}$. When viewing relations as sets it comes natural to form the *union* $R \cup S : X \leftrightarrow Y$ and *intersection* $R \cap S : X \leftrightarrow Y$ of two relations $R, S : X \leftrightarrow Y$ or to state the *inclusion* $R \subseteq S$. A lot of the expressive power of relation algebra is due to the possibility to express the *composition* $RS : X \leftrightarrow Y$

of two relations $R : X \leftrightarrow Z$ and $S : Z \leftrightarrow Y$. Its definition in predicate logic is that for all $x \in X$ and $y \in Y$ we have $(RS)_{x,y}$ iff there exists a $z \in Z$ such that $R_{x,z}$ and $S_{z,y}$. The *complementation* of a relation $R : X \leftrightarrow Y$ is denoted by $\overline{R} : X \leftrightarrow Y$ and corresponds to negation in predicate logic: for all x, y we have $\overline{R}_{x,y}$ iff $R_{x,y}$ does not hold.

By $\text{syq}(R, S) := \overline{R^{\mathsf{T}}\,\overline{S}} \cap \overline{\overline{R}^{\mathsf{T}} S} : Y \leftrightarrow Z$ the *symmetric quotient* of two relations $R : X \leftrightarrow Y$ and $S : X \leftrightarrow Z$ is defined. A specification in predicate logic is that $\text{syq}(R, S)_{y,z}$ iff for all x we have that $R_{x,y}$ iff $S_{x,z}$. In other words for all $y \in Y$ and $z \in Z$ we have $\text{syq}(R, S)_{y,z}$ iff the y-column of R equals the z-column of S. Additional properties of this construct can be found in [10].

2.2 Pairing and Related Constructions

The *pairing* (or *fork*) $[R, S] : Z \leftrightarrow X\times Y$ of two relations $R : Z \leftrightarrow X$ and $S : Z \leftrightarrow Y$ is defined by demanding for all $z \in Z$ and $u = \langle u_1, u_2\rangle \in X\times Y$ that $[R, S]_{z,u}$ iff R_{z,u_1} and S_{z,u_2}. It should be noted that throughout this paper pairs $u \in X\times Y$ are assumed to be of the form $\langle u_1, u_2\rangle$.

Using identity and universal relations of appropriate types, pairing allows to define the *projection relations* $\pi : X\times Y \leftrightarrow X$ and $\rho : X\times Y \leftrightarrow Y$ of the direct product $X\times Y$ as $\pi := [\mathsf{I}, \mathsf{L}]^{\mathsf{T}}$ and $\rho := [\mathsf{L}, \mathsf{I}]^{\mathsf{T}}$. Then the above definition implies for all $u \in X\times Y$, $x \in X$ and $y \in Y$ that $\pi_{u,x}$ iff $u_1 = x$ and $\rho_{u,y}$ iff $u_2 = y$. Also the *parallel composition* (or *product*) $R \parallel S : X\times X' \leftrightarrow Y\times Y'$ of two relations $R : X \leftrightarrow Y$ and $S : X' \leftrightarrow Y'$, such that $(R \parallel S)_{u,v}$ is equivalent to R_{u_1,v_1} and S_{u_2,v_2} for all $u \in X\times X'$ and $v \in Y\times Y'$, can be defined by means of pairing. We get the desired property if we define $R \parallel S := [\pi R, \rho S]$, where $\pi : X\times X' \leftrightarrow X$ and $\rho : X\times X' \leftrightarrow X'$ are the projection relations on $X\times X'$.

2.3 The Representation of Sets

In relation algebra sets can be modeled using *vectors*, which are relations v with $v = v\mathsf{L}$. For a vector the range is irrelevant and we therefore consider vectors $v : X \leftrightarrow \mathbf{1}$ with a specific singleton set $\mathbf{1} = \{\perp\}$ as range and omit the second subscript, i.e., write v_x instead of $v_{x,\perp}$. Such a vector can be considered as a Boolean matrix with exactly one column, i.e., as a Boolean column vector, and *represents* the subset $\{x \in X \mid v_x\}$ of X. A non-empty vector v is said to be a *point* if $vv^{\mathsf{T}} \subseteq \mathsf{I}$, i.e., v is *injective*. This means that it represents a single element. In the Boolean matrix model a point $v : X \leftrightarrow \mathbf{1}$ is a Boolean column vector in which exactly one entry is 1.

To model sets we will also apply *membership-relations* $\mathsf{M} : X \leftrightarrow 2^X$ on X and its powerset 2^X. These specific relations are defined by demanding for all $x \in X$ and $Y \in 2^X$ that $\mathsf{M}_{x,Y}$ iff $x \in Y$. Using BDDs as implementation of relations as in RelView, the number of BDD-nodes for M is linear in the cardinality of X. See [4] for details.

Given an injective function $\imath$ from Y to X, we may consider Y as a subset of X by identifying it with its image under $\imath$. If Y is actually a subset of X and

$\imath$ is given as relation of type $[Y \leftrightarrow X]$ such that $\imath_{y,x}$ iff $y = x$ for all $y \in Y$ and $x \in X$, then the vector $\imath^{\mathsf{T}}\mathsf{L} : X \leftrightarrow \mathbf{1}$ represents Y as subset of X in the sense above. To model sets, we will apply as third technique that the transition in the other direction is also possible, i.e., the generation of a relation $\mathrm{inj}(v) : Y \leftrightarrow X$ from the vector representation $v : X \leftrightarrow \mathbf{1}$ of $Y \subseteq X$ such that for all $y \in Y$ and $x \in X$ we have $\mathrm{inj}(v)_{y,x}$ iff $y = x$.

A combination of injective functions with membership-relations allows a *column-wise enumeration* of sets of subsets. More specifically, if $v : 2^X \leftrightarrow \mathbf{1}$ represents a subset $\mathfrak{S}$ of the powerset 2^X in the sense defined above, then for all $x \in X$ and $Y \in \mathfrak{S}$ we get the equivalence of $(\mathsf{M}\,\mathrm{inj}(v)^{\mathsf{T}})_{x,Y}$ and $x \in Y$. This means that $S := \mathsf{M}\,\mathrm{inj}(v)^{\mathsf{T}} : X \leftrightarrow \mathfrak{S}$ is the relation-algebraic specification of membership on $\mathfrak{S}$, or, using matrix terminology, the elements of $\mathfrak{S}$ are represented precisely by the columns of S. Furthermore, a little reflection shows for all $Y, Z \in \mathfrak{S}$ the equivalence of $Y \subseteq Z$ and $\overline{S^{\mathsf{T}}\,\overline{S}}_{Y,Z}$. Therefore, $\overline{S^{\mathsf{T}}\,\overline{S}} : \mathfrak{S} \leftrightarrow \mathfrak{S}$ is the relation-algebraic specification of set inclusion on $\mathfrak{S}$.

2.4 Extremal Elements of Orders and Lattices

Given a relation $R : X \leftrightarrow X$, the pair (X, R) is an ordered set iff $\mathsf{I} \subseteq R$ (reflexivity), $R \cap R^{\mathsf{T}} \subseteq \mathsf{I}$ (antisymmetry) and $RR \subseteq R$ (transitivity). In the following we may omit the set X when clear from context and simply refer to R as a partial order. When dealing with ordered sets, one typically investigates extremal elements. Based upon the vector representation of sets we will use the following relation-algebraic specifications taken from [10].

$$\mathrm{lel}(R, v) := v \cap \overline{\overline{R}\,v} \qquad \mathrm{gel}(R, v) := \mathrm{lel}(R^{\mathsf{T}}, v)$$
$$\mathrm{glb}(R, v) := \mathrm{gel}(R, \overline{\overline{R}\,v}\,) \qquad \mathrm{lub}(R, v) := \mathrm{glb}(R^{\mathsf{T}}, v)$$

If $R : X \leftrightarrow X$ is a partial order relation and Y a subset of X that is represented by the vector $v : X \leftrightarrow \mathbf{1}$, then $\mathrm{lel}(R, v) : X \leftrightarrow \mathbf{1}$ is empty iff Y does not have a least element and is a point that represents the least element of Y, otherwise. Similarly, $\mathrm{gel}(R, v) : X \leftrightarrow \mathbf{1}$ ($\mathrm{glb}(R, v) : X \leftrightarrow \mathbf{1}$ and $\mathrm{lub}(R, v) : X \leftrightarrow \mathbf{1}$, respectively) is either empty or a point that represents the greatest element (greatest lower bound and least upper bound, respectively) of Y,

If the second arguments of the above specifications are not vectors but "proper" relations with a non-singleton range, then the corresponding extremal elements are computed column-wisely. E.g., in the case of the first specification this means the following. For all $A : X \leftrightarrow Y$ we obtain $\mathrm{lel}(R, A) : X \leftrightarrow Y$ and, furthermore, for all $x \in X$ and $y \in Y$ that $\mathrm{lel}(R, A)_{x,y}$ iff the least element of $\{z \in X \mid A_{z,y}\}$ exists and equals x. Hence, for all $y \in Y$ the y-column of $\mathrm{lel}(R, A)$ is either empty or a point that represents (with respect to R) the least element of the set the y-column of A represents.

For a partial order $R : X \leftrightarrow X$ we also need the following specifications, which both are of type $[X \times X \leftrightarrow X]$:

$$\mathrm{Inf}(R) := [R, R]^{\mathsf{T}} \cap \overline{[R, R]^{\mathsf{T}}\,\overline{R}} \qquad \mathrm{Sup}(R) := \mathrm{Inf}(R^{\mathsf{T}})$$

In [2] it is shown that for all $u \in X \times X$ and $x \in X$ we have $\mathrm{Inf}(R)_{u,x}$ iff x is the greatest lower bound of u_1 and u_2 and $\mathrm{Sup}(R)_{u,x}$ iff x is the least upper bound of u_1 and u_2. Hence, $\mathrm{Inf}(R)$ and $\mathrm{Sup}(R)$ relation-algebraically specify the two lattice perations $\sqcap$ and $\sqcup$. As a consequence, an ordered set (X, R) constitutes a lattice $(X, \sqcup, \sqcap)$ iff $\mathsf{L} = \mathrm{Inf}(R)\mathsf{L}$ and $\mathsf{L} = \mathrm{Sup}(R)\mathsf{L}$, since the latter equations express that the two relations $\mathrm{Inf}(R)$ and $\mathrm{Sup}(R)$ are total (see [10]). Also complete lattices can easily be characterized by relation-algebraic means. If $R : X \leftrightarrow X$ is the partial order of a lattice $(X, \sqcup, \sqcap)$, then X is complete iff $\mathsf{L} = \mathsf{L}\,\mathrm{glb}(R, \mathsf{M})$ or, equivalently, iff $\mathsf{L} = \mathsf{L}\,\mathrm{lub}(R, \mathsf{M})$, where $\mathsf{M} : X \leftrightarrow 2^X$ is the membership relation.

3 Computing and Visualizing Closure Objects

In the introduction we have mentioned several concepts which we refer to as "closure objects". In the literature, all of these notions are usually defined on powersets; see e.g., [6]. But with the exception of dependency relations the restriction to such a specific class of lattices is not necessary. We therefore prefer to define all closure objects on more general ordered structures. In this section, we develop relation-algebraic specifications for recognizing, computing and transforming closure objects on complete lattices. The specifications will be algorithmic and, hence, can directly be translated into RelView-code. Since RelView only allows to treat relations on finite sets, we assume for the developments finite lattices if this is advantageous.

3.1 Closure Systems

Assume $(X, \sqcup, \sqcap)$ to be a complete lattice. Then $S \subseteq X$ is called a *closure system* (or a Moore family) of X if it is closed under arbitrary least upper bounds, that is, for all $X \subseteq S$ we have $\bigsqcap X \in S$. In the case of a finite carrier set X this second order definition is obviously equivalent to the two requirements 1) that $\top \in S$, where $\top$ denotes the greatest element of the lattice, and 2) that for all $x, y \in S$ also $x \sqcap y \in S$. The next theorem provides the transformation of this first-order specification to relation algebra.

Theorem 3.1.1 *Assume $R : X \leftrightarrow X$ to be the partial order of a finite lattice $(X, \sqcup, \sqcap)$ and let $S \subseteq X$ be represented by the vector $s : X \leftrightarrow \mathbf{1}$. Then S is a closure system of X iff* $\mathrm{gel}(R, \mathsf{L}) \subseteq s$ *and* $[s^\mathsf{T}, s^\mathsf{T}]^\mathsf{T} \subseteq \mathrm{Inf}(R)\, s$.

The formulae of Theorem 3.1.1 can immediately be translated into the programming language of the computer algebra system RelView. Hence, given a partial order $R : X \leftrightarrow X$ of a lattice and a vector $s : X \leftrightarrow \mathbf{1}$, the tool can be used to test whether s represents a closure system. Note that in Theorem 3.1.1 the type of s^T is $[\mathbf{1} \leftrightarrow X]$ and, thus, the type of $[s^\mathsf{T}, s^\mathsf{T}]$ is $[\mathbf{1} \leftrightarrow X \times X]$. This is advantageous for the implementation in RelView, which is based on BDDs. In general, the transposition of a relation requires that a new BDD has to be computed from the

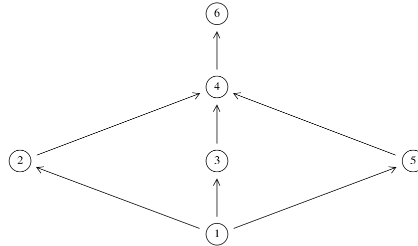


Fig. 1. Hasse-diagram of the partial order of a lattice with 6 elements

old one by exchanging the variables encoding the domain with those encoding the range. But in the case of a relation with domain or range $\mathbf{1}$ this process can be omitted since the BDD of the relation and its transpose coincide [4].

Having specified a single closure system within relation algebra, we turn to specify the set $\mathfrak{S}(X)$ of all closure systems of X as a subset of the powerset via a vector of type $[2^X \leftrightarrow \mathbf{1}]$. In the following theorem $\mathsf{M} : X \leftrightarrow 2^X$ is a membership relation, $\pi, \rho : X \times X \leftrightarrow X$ are the projection relations of $X \times X$, the left L has type $[X \leftrightarrow \mathbf{1}]$ and the remaining L is of type $[\mathbf{1} \leftrightarrow X \times X]$.

Theorem 3.1.2 *Let again $R : X \leftrightarrow X$ be the partial order of a finite lattice $(X, \sqcup, \sqcap)$. Then* $\mathrm{cls}(R) := (\mathrm{gel}(R, \mathsf{L})^\mathsf{T} \mathsf{M} \cap \overline{\mathsf{L}(\pi \mathsf{M} \cap \rho \mathsf{M} \cap \overline{\mathrm{Inf}(R)\,\mathsf{M}}\,)}\,)^\mathsf{T} : 2^X \leftrightarrow \mathbf{1}$ *is a vector-representation of the set $\mathfrak{S}(X)$ of all closure systems of X.*

Again, the transpositions occurring in the definition of $\mathrm{cls}(R)$ are motivated by the aim to obtain an efficient RelView program. Stating the formula in the given way, the relations affected during program execution are all of domain $\mathbf{1}$. If, in contrast, we would not have simplified the result by applying the rule $R^\mathsf{T} S^\mathsf{T} = (SR)^\mathsf{T}$ the resulting program would be less efficient and not scale anymore.

A significant fact about the set $\mathfrak{S}(X)$ is that it is itself a closure system of the powerset lattice $(2^V, \cup, \cap)$. Hence, it forms a complete lattice with intersection as greatest lower bound operation and set inclusion as partial order. A relation-algebraic specification of the partial order of $\mathfrak{S}(X)$ is rather simple. Using the technique described in Section 2.3, by

$$\mathrm{ClSys}(R) := \mathsf{M}\,\mathrm{inj}(\mathrm{cls}(R))^\mathsf{T} : X \leftrightarrow \mathfrak{S}(X)$$

the set $\mathfrak{S}(X)$ is enumerated by column and we immediately obtain from $\mathrm{ClSys}(R)$ that the set inclusion on $\mathfrak{S}(X)$ can be specified as

$$\mathrm{ClLat}(R) := \overline{\mathrm{ClSys}(R)^\mathsf{T}\,\overline{\mathrm{ClSys}(R)}} : \mathfrak{S}(X) \leftrightarrow \mathfrak{S}(X).$$

The number of different closure systems of a finite lattice grows very rapidly, see [6] for the numbers in the case of powersets 2^X up to $|X| = 6$. Therefore, a visualization of the according lattice is useful for rather small examples, only.

Example 3.1.1. Figure 1 contains a picture of the Hasse-diagram of the partial order of a lattice X^* with 6 elements $x_1, \ldots . x_6$. The picture demonstrates the

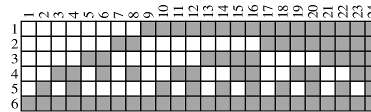


Fig. 2. Closure systems of the partial order of Figure 1

support in RelView to depict relations as directed graphs. Each vertex with label n represents the lattice element x_n, $1 \leq n \leq 6$.

Stating the two relation-algebraic specifications $\mathrm{ClSys}(R)$ and $\mathrm{ClLat}(R)$ above programs in the programming language of RelView, we computed that 24 of the 64 subsets of X^* are closure systems. The result of this computation is shown in Figure 2. There the 24 subsets are enumerated by column in a 6×24 Boolean matrix. In the picture a filled square denotes a 1-entry and an empty square a 0-entry. If we denote the closure system represented by column i with S_i, $1 \leq i \leq 24$, then, e.g., the first column represents the closure system $S_1 = \{x_6\}$ consisting of the greatest lattice element only, the second column represents the closure system $S_2 = \{x_5, x_6\}$, the third column represents the closure system $S_3 = \{x_4, x_6\}$ and the last column represents the closure system $S_{24} = X^*$ consisting of all lattice elements.

Finally, Figure 3 shows the Hasse-diagram of the lattice $\mathfrak{S}(X^*)$, where the vertex with label i corresponds to the closure system S_i, $1 \leq i \leq 24$, and an arrow denotes set inclusion. From Figure 2 it follows that the n-th layer of the graph exactly contains the vertices corresponding to the closure systems with cardinality n, $1 \leq n \leq 6$.

3.2 Closure Operations

For a given ordered set (X, R) a *closure operation* is a function $C : X \to X$ which is 1) extensive, 2) monotone, and 3) idempotent. Note that it is not necessary to restrict such operations to sets as arguments. In the next theorem we provide

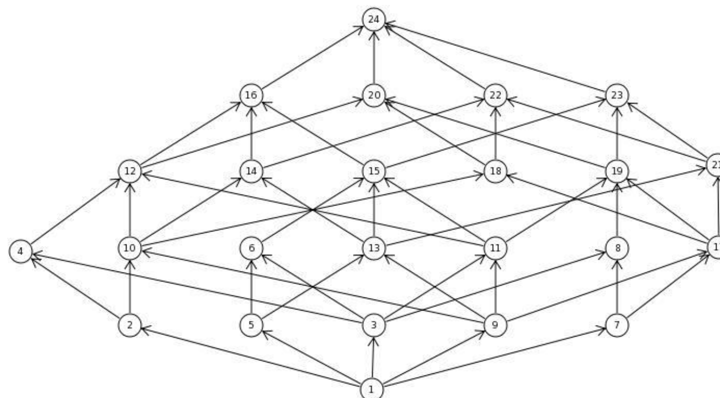


Fig. 3. Hasse-diagram of the lattice $\mathfrak{S}(X^*)$

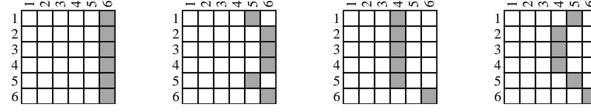


Fig. 4. Closure operations $C_1 \dots C_4$ of $\mathfrak{O}(X^*)$

a relation-algebraic characterization of closure operations of (X, R) as specific relations of type $[X \leftrightarrow X]$. Again the given formulae directly lead to a RelView-program for recognizing closure operations.

Theorem 3.2.1 *Given a partial order $R : X \leftrightarrow X$, a relation $C : X \leftrightarrow X$ is a closure operation of the ordered set (X, R) iff $C\bar{\mathsf{I}} = \overline{C}$, $C \subseteq R$, $R \subseteq CRC^{\mathsf{T}}$ and $CC \subseteq C$.*

On complete lattices $(X, \sqcup, \sqcap)$ there is a well-known one-to-one correspondence between the set $\mathfrak{O}(X)$ of all closure operations and the set $\mathfrak{S}(X)$ of all closure systems. The closure system corresponding to the closure operation $C \in \mathfrak{O}(X)$ is the set of all fixed points of C. Conversely, the closure operation corresponding to the closure system $S \in \mathfrak{S}(X)$ is such that $x \in X$ is mapped to $\bigsqcap\{z \in S \mid R_{x,z}\}$, where $R : X \leftrightarrow X$ is the partial order of the lattice. The following theorem states how these correspondences can be formulated as a pair of relation-algebraic specifications. To this end, we identify the subsets of X with their representation as vectors from $[X \leftrightarrow \mathbf{1}]$. This enables us to consider $\mathfrak{S}(X)$ to as a subset of the powerset $2^{[X \leftrightarrow \mathbf{1}]}$, and $\mathfrak{O}(X)$ as a subset of $[X \leftrightarrow X]$.

Theorem 3.2.2 *Let $R : X \leftrightarrow X$ be the partial order of a lattice $(X, \sqcup, \sqcap)$. Then, for all $C \in \mathfrak{O}(X)$ the vector $\mathrm{CloToCls}(C) := (C \cap \mathsf{I})\mathsf{L} : X \leftrightarrow \mathbf{1}$ represents the set of all fixed points of C and for all $s \in \mathfrak{S}(X)$ the relation $\mathrm{ClsToClo}(s) := \mathrm{glb}(R, s\mathsf{L} \cap R^{\mathsf{T}})^{\mathsf{T}} : X \leftrightarrow X$ fulfills for all $x, y \in X$ that $\mathrm{ClsToClo}(s)_{x,y}$ iff $y = \bigsqcap\{z \in V \mid s_z \wedge R_{x,z}\}$.*

The two functions $\mathrm{CloToCls} : \mathfrak{O}(X) \to \mathfrak{S}(X)$ and $\mathrm{ClsToClo} : \mathfrak{S}(X) \to \mathfrak{O}(X)$ of Theorem 3.2.2 are order-*reversing* (antiton) with respect to set inclusion for closure systems and the pointwise ordering of functions. The latter order on functions can be specified in relation algebra as $C_1 \leq C_2$ iff $C_1 \subseteq C_2R^{\mathsf{T}}$ since $C_1 \subseteq C_2R^{\mathsf{T}}$ says that for all $x, y \in X$ if C_1 maps x to y then C_2 maps x to a $z \in X$ with $R_{y,z}$.

Example 3.2.1. As the functions of Theorem 3.2.2 are order-reversing (antiton), a transposition of the Hasse-diagram in Figure 3 yields the Hasse-diagram of the lattice $\mathfrak{O}(X^*)$. Accordingly, in the resulting graph the vertex with label i represents the closure operation C_i corresponding to the closure system S_i for $1 \leq i \leq 24$ and Figure 4 depicts the relations $C_1, \dots, C_4$. The function C_1 maps all elements to the greatest lattice element. It is the greatest closure operation with respect to the pointwise ordering. The least closure operation is the identity relation I and corresponds to the greatest closure system S_{24}.

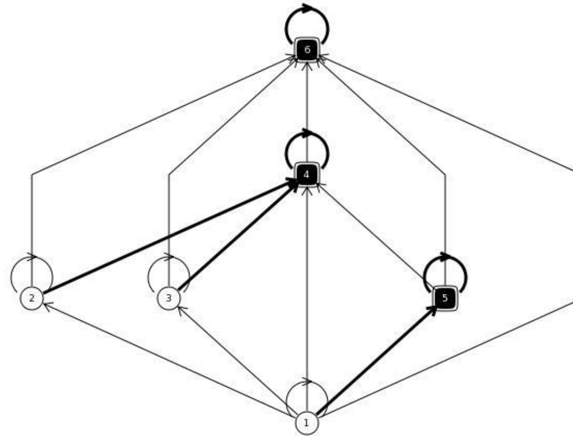


Fig. 5. Lattice X^*, emphasizing closure system S_4 and closure operation C_4

Figure 5 shows the partial order relation of the lattice X^* as directed graph, where the vertices of the closure system S_4 are emphasized as black squares and the arcs corresponding to the pairs of the closure operation C_4 are drawn boldface. Two of the three properties of closure operations can immediately be verified by examining the picture. The operation C_4 is extensive, since each arc of C_4 is an arc of the graph. It is idempotent, since each C_4-path leads into a loop over at most one non-loop edge. Monotonicity of C_4 can be recognized by pointwise comparisons. The picture also clearly visualizes that each element of the lattice is either a fixed point of C_4, i.e., contained in the corresponding closure system S_4, or is mapped to the least element of S_4 above it.

The *topological closure operations* of a lattice $(X, \sqcup, \sqcap)$ distribute over the $\sqcup$-operation and form an important subclass of $\mathfrak{O}(X)$. If the lattice X is finite, the corresponding closure systems are precisely the sublattices of X which contain the greatest element of X. Assuming $R : X \leftrightarrow X$ as partial order of X, a calculation shows that $C \in \mathfrak{O}(X)$ is topological iff $(C \parallel C)\,\mathrm{Sup}(R) = \mathrm{Sup}(R)\,C$. We have transformed this equation into RelView-code and computed for the above example lattice X^* that exactly 20 out of the 24 closure operations are topological. The four exceptions are C_{14}, C_{18}, C_{21} and C_{22}.

3.3 Full Implicational Systems and Join-Congruences

The origin of full implicational systems is relational database theory, where they are called families of functional dependencies (see e.g., [7]). In [6] full implicational systems are defined on powersets by a variant of the well known Armstrong axioms which require for the relation, written as arrow, and for all sets A, B, C, D that 1) if $A \to B$ and $B \to C$ then $A \to C$, 2) if $A \supseteq B$ then $A \to B$ and 3) if $A \to B$ and $C \to D$ then $A \cup C \to B \cup D$.

We generalize this description to finite (complete) lattices $(X, \sqcup, \sqcap)$ with partial order $R : X \leftrightarrow X$. In this sense, a *full implicational system* on X is a

relation $F : X \leftrightarrow X$ that is 1) transitive, 2) contains R^{T} and 3) for all $x, y, x', y' \in X$ it holds that $F_{x,x'}$ and $F_{y,y'}$ imply $F_{x \sqcup y, x' \sqcup y'}$. As a side remark we note that axiom 3) could be generalized to arbitrary least upper bounds, i.e., arbitrary complete lattices. The resulting relation-algebraic formulation would be that 3') for all subrelations $D \subseteq F$ we have $vw^{\mathsf{T}} \subseteq F$, where $v := \mathrm{lub}(R, D\mathsf{L})$ specifies the least upper bound of all first components of pairs of D and $w := \mathrm{lub}(R, D^{\mathsf{T}}\mathsf{L})$ does the same for the second components. But as we restrict ourselves to finite relations, the following theorem considers the first version of the axiom only.

Theorem 3.3.1 *Given $R : X \leftrightarrow X$ as partial order of a finite lattice $(X, \sqcup, \sqcap)$, $F : X \leftrightarrow X$ is a full implicational system of X iff $FF \subseteq F$, $R^{\mathsf{T}} \subseteq F$ and $F \parallel F \subseteq \mathrm{Sup}(R)\, F\, \mathrm{Sup}(R)^{\mathsf{T}}$.*

If full implicational systems are ordered by inclusion, then the complete lattice induced by $(\mathfrak{O}(X), \leq)$ is isomorphic to the complete lattice induced by $(\mathfrak{F}(X), \subseteq)$, where $\mathfrak{F}(X)$ denotes the set of all full implicational systems of $(X, \sqcup, \sqcap)$. One direction of this isomorphism is given by mapping $C \in \mathfrak{O}(X)$ to the full implicational system $F \in \mathfrak{F}(X)$ that consists of all pairs $\langle x, y\rangle \in X \times X$ with $R_{y,C(x)}$. The converse direction is obtained by mapping $F \in \mathfrak{F}(X)$ to the closure operation $C \in \mathfrak{O}(X)$ such that $C(x) = \bigsqcup\{z \in X \mid F_{x,z}\}$ for all $x \in X$. The following theorem yields these correspondences formulated as a pair of relation-algebraic specifications.

Theorem 3.3.2 *Let $R : X \leftrightarrow X$ be the partial order of a lattice $(X, \sqcup, \sqcap)$. Then, for all $C \in \mathfrak{O}(X)$ the relation $\mathrm{CloToFis}(C) := CR^{\mathsf{T}} : X \leftrightarrow X$ fulfills for all $x, y \in X$ that $\mathrm{CloToFis}(C)_{x,y}$ iff $R_{y,C(x)}$, and, conversely, for all $F \in \mathfrak{F}(X)$ the relation $\mathrm{FisToClo}(F) := \mathrm{lub}(R, F^{\mathsf{T}})^{\mathsf{T}} : X \leftrightarrow X$ fulfills for all $x, y \in X$ that $\mathrm{FisToClo}(F)_{x,y}$ iff $y = \bigsqcup\{z \in X \mid F_{x,z}\}$.*

There is a very close relation between full implicational systems and join-congruence relations, which are generalizations of lattice congruences. Given a lattice $(X, \sqcup, \sqcap)$, a relation $J : X \leftrightarrow X$ is a *join-congruence* of X iff it is an equivalence relation and, in addition for all $x, y, z \in X$ from $J_{x,y}$ it follows that $J_{x \sqcup z, y \sqcup z}$. How to specify equivalence relations with relation-algebraic means is well-known; see e.g., [10]. The remaining requirement on join-congruences holds for J iff $J \parallel \mathsf{I} \subseteq \mathrm{Sup}(R)\, J\, \mathrm{Sup}(R)^{\mathsf{T}}$. This leads to the following result.

Theorem 3.3.3 *Let $R : X \leftrightarrow X$ be the partial order of a lattice $(X, \sqcup, \sqcap)$. Then $J : X \leftrightarrow X$ is a join-congruence of X iff $\mathsf{I} \subseteq J$, $J = J^{\mathsf{T}}$, $JJ \subseteq J$ and $J \parallel \mathsf{I} \subseteq \mathrm{Sup}(R)\, J\, \mathrm{Sup}(R)^{\mathsf{T}}$.*

In the case of a finite lattice $(X, \sqcup, \sqcap)$ there is a one-to-one correspondence between the set $\mathfrak{O}(X)$ of all closure operations of X and the set $\mathfrak{J}(X)$ of all join-congruences of X which again establishes a lattice isomorphism wrt. the lattices induced by the ordered sets $(\mathfrak{O}(X), \leq)$ and $(\mathfrak{J}(X), \subseteq)$. The join-congruence J associated with the closure operation $C \in \mathfrak{O}(X)$ is the kernel of the function C, i.e., we have for all $x, y \in X$ that $J_{x,y}$ iff $C(x) = C(y)$. Relation-algebraically

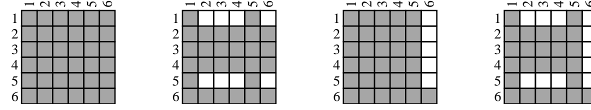


Fig. 6. Full implicational systems F_1 to F_4

this means that $J = \text{CloToJc}(C)$, where $\text{CloToJc}(C) := CC^{\mathsf{T}} : X \leftrightarrow X$. In the reverse direction, the closure operation C is obtained from the join-congruence $J \in \mathfrak{J}(X)$ as in the case of full implicational systems, i.e., by mapping each element $x \in X$ to $\bigsqcup\{z \in X \mid J_{x,z}\}$. Using Theorem 3.3.2 we get $C = \text{JcToClo}(J)$ as $\text{JcToClo}(J) := \text{lub}(R, J)^{\mathsf{T}} : X \leftrightarrow X$ where $R : X \leftrightarrow X$ is the partial order of the finite lattice $(X, \sqcup, \sqcap)$.

Example 3.3.1. In Figure 6 the four full implicational systems F_1 to F_4 of our running example are shown as four RelView-pictures, where the relation F_i is the value of the specification $\text{CloToFis}(C_i)$ with the closure operations C_i from Figure 4, $1 \leq i \leq 4$.

For our running example we already know that exactly 24 equivalence relations $J_i = \text{CloToJc}(C_i)$, $1 \leq i \leq 24$, on X^* are join-congruences. Figure 7 shows the relations J_1 to J_4. Each column (or row) directly corresponds to a congruence class of the respective relation. In addition we used RelView to test which of the 24 join-congruences are also meet-congruences. Analogously, a meet-congruence M satisfies for all $x, y, z \in X$ that $M_{x,y}$ implies $M_{x \sqcap z, y \sqcap z}$. We obtained four positive answers: J_1, J_3, J_{22} (with the classes $\{x_1\}, \{x_2\}, \{x_3\}, \{x_4, x_6\}, \{x_5\}$) and $\mathsf{I} = J_{24}$. The relation J_2, for example, is not a meet-congruence, since $x = z = x_2$ and $y = x_3$ is one of the 12 triples such that $\langle x, y\rangle$ is in J_2 but $\langle x \sqcap z, y \sqcap z\rangle$ is not in J_2.

3.4 Dependency Relations

In [1] Aumann introduced certain relations to formalize the essential properties of a "contact" between objects and sets of objects. His motivation was to obtain an access to topology which is more suggestive for beginners than the ones provided by "traditional" axiom systems. If we formulate Aumann's original definition in our notation, then a relation $D : X \leftrightarrow 2^X$ is a *contact* if 1) for all $x \in X$ and $Y, Z \in 2^X$ we have $D_{x,\{x\}}$, that 2) from $D_{x,Y}$ and $Y \subseteq Z$ it follows $D_{x,Z}$, and that 3) from $D_{x,Y}$ it follows $D_{x,Z}$ if $D_{y,Z}$ holds for all $y \in Y$.

Obviously, demands 1) and 2) are equivalent to the fact that $x \in Y$ implies $D_{x,Y}$ for all $x \in X$ and $Y \in 2^X$. Hence, Aumann's contacts are exactly the

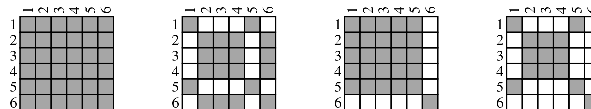


Fig. 7. Join congruences J_1 to J_4

dependency relations in the sense of [6]. In the following theorem, we present relation-algebraic versions of the two axioms given in [6].

Theorem 3.4.1 *Let* $\mathsf{M} : X \leftrightarrow 2^X$ *be a membership-relation. Then a relation* $D : X \leftrightarrow 2^X$ *is a dependency relation iff* $\mathsf{M} \subseteq D$ *and* $D\,\overline{\mathsf{M}^\mathsf{T}\,\overline{D}} \subseteq D$.

In [1] a one-to-one correspondence between the set $\mathfrak{O}(2^X)$ of all closure operations of $(2^X, \subseteq)$ and the set $\mathfrak{D}(X)$ of all contacts of type $[X \leftrightarrow 2^X]$ is established, which is also mentioned in [6] for dependency relations. The relation $D \in \mathfrak{D}(X)$ corresponding to $C \in \mathfrak{O}(2^X)$ is for all $x \in X$ and $Y \in 2^X$ given by $D_{x,Y}$ iff $x \in C(Y)$. Conversely, the closure operation associated with $D \in \mathfrak{D}(X)$ maps $Y \in 2^X$ to $\{x \in X \mid D_{x,Y}\}$. The next theorem contains corresponding specifications in relation algebra.

Theorem 3.4.2 *Assume* $\mathsf{M} : X \leftrightarrow 2^X$ *to be a membership-relation. Then, for all* $C \in \mathfrak{O}(2^X)$ *the relation* $\mathrm{CloToDep}(C) := \mathsf{M}C^\mathsf{T} : X \leftrightarrow 2^X$ *fulfills for all* $x \in X$ *and* $Y \in 2^X$ *that* $\mathrm{CloToDep}(C)_{x,Y}$ *iff* $x \in C(Y)$*, and, conversely, for all* $D \in \mathfrak{D}(X)$ *the relation* $\mathrm{DepToClo}(D) := \mathrm{syq}(D, \mathsf{M}) : 2^X \leftrightarrow 2^X$ *fulfills for all* $Y \in X$ *that* $\mathrm{DepToClo}(D)(Y) = \{x \in X \mid D_{x,Y}\}$.

In [1] Aumann mentions that his relations may also be used to investigate the notion of a contact in sociology or political science. For this, it is frequently necessary to replace M by a relation $M : X \leftrightarrow G$ with the interpretation "individual x is a member of a group g of individuals" of $M_{x,g}$. If $\mathrm{syq}(M, M) = \mathsf{I}$, then (G, R) is an ordered set, where $R := \overline{M^\mathsf{T}\,\overline{M}}$. In this general setting the one-to-one correspondence between closure operations and contacts is lost. It can only be shown that there is an order embedding from the set of closure operations to the set of these generalized contacts.

4 Conclusion

Closure systems and closure operations play an important role in both mathematics and computer science. Moreover, the literature contains many examples for concepts employed in practice which could be proven to be isomorphic to special closure systems. In this work we have presented relation-algebraic formulations of the connections between closure systems on the one hand and closure operations, full implication systems, join-congruences, and dependency relations on the other hand. The resulting algebraic representations are very compact. We have demonstrated that the formulas can directly be used to compute the transformation between concepts, e.g., transform a given finite dependency relation to the corresponding closure operation. Each of the definitions can also be used to efficiently test given relations for conformance, e.g., to compute whether a given relation is a dependency relation or not. We have used the computer algebra system RelView to compute both transformations and tests. As shown by examples, the system can also be used to visualize the results either as graphs or as Boolean matrices, whatever is more appropriate to the case.

A final word about scalability. We have taken care that the presented tests and transformations fit well into the setting of relations implemented using BDDs. As a consequence all of the resulting programs scale well and are also applicable to big relations with ten thousands of elements. The only exception is the enumeration of all possible closure systems on a given set provided with Theorem 3.1.2. As shown in [6] the number of such systems grows too rapidly to be subject to an efficient complete enumeration.

References

1. Aumann G.: Contact relations (in German). Bayerische Akademie der Wissenschaften, Math.-Nat. Klasse Sitzungsberichte 1970, 67-77 (1970).
2. Berghammer R.: Solving algorithmic problems on orders and lattices by relation algebra and RelView. In: Gansha V.G. et al. (eds.): Computer Algebra in Scientific Computing, LNCS 4194, Springer, 49-63 (2006).
3. Berghammer R., Braßel B.: Computing and visualizing closure objects using relation algebra and RelView. In: Gerdt V.P. et al. (eds.): Computer Algebra in Scientific Computing, LNCS 5743, Springer, 29-44 (2009).
4. Berghammer R., Leoniuk B., Milanese U.: Implementation of relation algebra using binary decision diagrams. In: de Swart H. (ed.): Relational Methods in Computer Science, LNCS 2561, Springer, 241-257 (2002).
5. Berghammer R., Neumann F.: RelView– An OBDD-based Computer Algebra system for relations. In: Gansha V.G. et al. (eds.): Computer Algebra in Scientific Computing, LNCS 3718, Springer, 40-51 (2005).
6. Caspard N., Monjardet B.: The lattices of closure systems, closure operators, and implicational systems on a finite set: a survey. Discr. Appl. Math. 127, 241 - 269 (2003).
7. Demetrovics J., Lipkin L.O., Muchnik J.B.: Functional dependencies in relational databases: a lattice point of view. Discr. Appl. Math. 40, 155-185 (1992).
8. Doignon J.P., Falmagne J.C.: Knowledge spaces. Springer (1999).
9. Ganter B., Wille R.: Formal concept analysis, Mathematical foundations. Springer (1998).
10. Schmidt G., Ströhlein T.: Relations and graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theor. Comp. Sci. Springer (1993).
11. Tarski A.: On the calculus of relations. J. Symb. Logic 6, 73-89 (1941).

Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine

Walter Binder, Alex Villazón, Danilo Ansaloni, and Philippe Moret

Faculty of Informatics, University of Lugano, Switzerland
`firstname.lastname@usi.ch`

Abstract. Many software engineering tools for the Java Virtual Machine that perform some form of dynamic program analysis, such as profilers or debuggers, are implemented with low-level bytecode instrumentation techniques. While program manipulation at the bytecode level is very flexible, because the possible bytecode transformations are not restricted, tool development is tedious and error-prone. Specifying bytecode instrumentations at a higher level using aspect-oriented programming (AOP) is a promising alternative in order to reduce tool development time and cost. However, prevailing AOP frameworks lack some features that are essential for certain dynamic analyses. In this paper, we focus on two common shortcomings in AOP frameworks with respect to the development of aspect-based tools – (1) the lack of mechanisms for passing data between woven advices in local variables, and (2) the support for user-defined static analyses at weaving time. We introduce @J, an annotation-based AOP language and weaver that integrates support for these two features. We illustrate the benefits of the proposed features with an example.

Keywords. Aspect-oriented programming, aspect weaving, local variables, analysis at weaving time, bytecode instrumentation, dynamic program analysis, profiling, debugging, Java Virtual Machine

1 Introduction

Bytecode instrumentation techniques are widely used for building software engineering tools that perform some dynamic program analysis, such as profilers [16, 15, 21, 8, 26], memory leak detectors [41, 27], data race detectors [11, 12], or testing tools that preserve the conditions that caused a crash [4].[1]

Java supports bytecode instrumentation using native code agents through the Java Virtual Machine Tool Interface (JVMTI) [35], as well as portable bytecode instrumentation through the `java.lang.instrument` API. Several bytecode engineering libraries have been developed, such as BCEL [36], ASM [28], Javassist [13], or Soot [37], to mention some of them.

[1] In this paper we only consider program transformations that insert bytecode, but do not alter or delete existing bytecode in a program.

However, because of the low-level nature of bytecode and of bytecode engineering libraries, the implementation of new instrumentation tools can be difficult and error-prone, often requiring high development and testing effort. For example, a frequent mistake is the incorrect update of exception handler tables, which may result in instrumented code that passes many tests, but may later fail under particular conditions. As another drawback of low-level instrumentation techniques, the resulting software engineering tools are often complex and difficult to maintain and to extend.

Aspect-oriented programming (AOP) [23] enables the specification of cross-cutting concerns in applications, avoiding related code that is scattered throughout methods, classes, or components. Traditionally, AOP has been used for disposing of "design smells", such as needless repetition, and for improving maintainability of applications. AOP has also been successfully applied to the development of software engineering tools, such as profilers, debuggers, or testing tools [30, 6, 40], which in many cases can be specified as aspects[2] in a concise manner. Hence, in a sense, AOP can be regarded as a versatile approach for specifying certain program instrumentations at a high level, hiding low-level implementation details, such as bytecode manipulation, from the programmer.

However, current AOP frameworks have not been especially designed for the implementation of instrumentation-based software engineering tools. Some important features are missing, limiting the program instrumentations that can be expressed as aspects. We found the following two features, which are not supported by prevailing AOP frameworks such as AspectJ [22], essential in various case studies where we have tried applying AOP for recasting instrumentation-based software engineering tools as aspects.

- Data passing between advices that are woven into the same method using local variables. In many instrumentation-based tools, local variables are allocated to pass data between different instrumentation sites in the code. While AspectJ's `around` advice allows passing data generated by code inserted before a join point to code inserted after a join point, it is not possible to pass data in local variables between different join points, such as from a "`before call`" advice to an "`after execution`" advice.
- Execution of custom analysis code, which only depends on static information, at weaving time. Many instrumentation-based tools perform some specific analysis to determine whether and how a particular join point shall be instrumented. For example, the listener latency profiler LiLa [21] analyses the class hierarchy to determine whether an invoked method is declared in a listener interface; only in that case the method invocation is profiled. While AspectJ lacks support for user-defined analyses at weaving time, some other AOP frameworks, such as SCoPE [3], provide such features (see Section 5 for details).

[2] Aspects specify *pointcuts* to intercept certain points in the execution of programs (so-called *join points*), such as method calls, fields access, etc. *Advices* are executed *before*, *after*, or *around* the intercepted join points. Advices have access to contextual information of the join points.

In order to provide these missing features in an AOP framework, we are designing @J (Aspect Tools in Java), an annotation-based aspect language and weaver, especially intended for easing the implementation of instrumentation-based software engineering tools. @J supports many AspectJ constructs (in AspectJ's annotation version, which we will call @AspectJ in this paper), and the implementation of the @J weaver reuses the code of the @AspectJ weaver as much as possible. In this paper, we focus on the new constructs offered by @J that are not available in @AspectJ.

In @J, instrumentations are expressed as *code snippets*[3] which are woven at bytecode positions specified by the snippet programmer. By default, snippets are inlined in the woven code. @J supports *invocation-local variables*, allowing snippets that are woven into the same method body to pass data in local variables [38]. @J snippets may access context information, such as static or dynamic join point instances, in the same way as in @AspectJ.

@J supports *executable snippets*, allowing the expression of custom static analyses that are executed at weaving time. An executable snippet may only access static context information. By storing values in invocation-local variables, an executable snippet can pass the results of a static analysis to other inlined snippets. Apart from writing to invocation-local variables, executable snippets must not have any side effects. An executable snippet is woven by inserting a bytecode sequence that assigns the values to invocation-local variables that the snippet has generated upon execution at weaving time. As snippets can be composed, the code inserted in a woven method at a particular bytecode position may consist of an arbitary sequence of inlined and executable snippets. Hence, custom static analyses can be embedded within inlined code snippets. For a more detailed discussion of @J features, we refer to [9].

This paper is structured as follows: Section 2 summarizes the design goals underlying @J. Section 3 discusses the distinguishing language features of @J. Section 4 gives an example @J program, illustrating the use of @J's special features. Section 5 discusses related work, and Section 6 concludes this paper.

2 Design Goals

In this section we summarize the design goals underlying @J.

- **Expressiveness:** @J is designed to allow the expression of a wide range of instrumentation-based software engineering tools. We have explored a large variety of case studies in the profiling and debugging domains in order to determine the necessary features. Examples include the dynamic metrics collector *J [16], the NetBeans Profiler [27], the latency listener profiler LiLa [21], the testing tool ReCrash [4], and the Eclipse plugin Senseo that

[3] In @J we always use the term "snippet" instead of "advice" for the code to be executed at an intercepted join point, because we found it more intuitive for programming instrumentation-based software engineering tools.

collects various dynamic metrics and runtime type information [33]. @J allows recasting the considered case studies as compact snippets that can be easily extended.
- **Efficiency:** @J shall enable the construction of efficient software engineering tools that offer the same level of runtime performance as tools programmed with low-level bytecode engineering libraries.
- **Portability and compatibility:** @J is implemented in pure Java. Snippets may be implemented in pure Java, too. Hence, snippet-based tools can be run in any standard Java Virtual Machine (JVM) (JDK 1.5 or higher). This is important, as we do not want to constrain tool users to employ a particular JVM. Snippet-based tools can be integrated into the users' preferred software development environment.
- **Full method coverage:** For many instrumentation-based dynamic analysis tools, such as profilers or memory leak detectors, it is essential that the instrumentation covers all methods executing in the JVM (which have a bytecode representation), including methods in dynamically generated or loaded classes, as well as in the Java class library. In addition, in some cases it is desirable to intercept also the execution of native methods. To ensure full method coverage, @J is based on the FERRARI framework[4] [7], which is also the basis of the MAJOR aspect weaver [39, 40].[5] Optionally, FERRARI can make use of native method prefixing, offered by the JVMTI [35] (which however requires JDK 1.6 or higher), in order to wrap native methods with bytecode versions that are amenable to snippet weaving.

3 @J Features

In this section, we firstly summarize the features of @AspectJ that are also supported in @J, and secondly explain the new features of @J that are complementary to @AspectJ.

3.1 Supported @AspectJ Features

@J supports @AspectJ pointcuts, as well as `before` and `after` advices. Static and dynamic join points are supported in the same way as in @AspectJ.

@J does not support non-singleton aspect instances using `per*` clauses (e.g., per-object or per-control flow aspect association), because @J snippets are either inlined or executed at weaving time.

@AspectJ's `around` advice is not supported in @J. The @AspectJ weaver implements the `around` advice by inserting wrapper methods in woven classes [20], which can cause problems when weaving the Java class library. For instance,

[4] `http://www.inf.usi.ch/projects/ferrari/`

[5] FERRARI prevents the execution of inserted code during JVM bootstrapping. That is, during the bootstrapping phase, @J snippets are not executed. However, full method coverage is guaranteed for the whole execution of a program's main thread, and for all threads spawned by the program.

in Sun's HotSpot JVMs there is a bug that limits the insertion of methods in `java.lang.Object`.[6] Moreover, wrapping certain methods in the Java class library breaks stack introspection in many recent JVMs, including Sun's HotSpot JVMs and IBM's J9 JVM [26, 40]; usually, there is no public documentation indicating those methods in the class library that must not be wrapped. Hence, the use of `around` advices would compromise weaving with full method coverage in many common, state-of-the-art JVMs. Nonetheless, with the aid of invocation-local variables, it is possible to emulate a common use of `around` advices as a combination of `before` and `after` advices.

Static cross-cutting (inter-type declarations) [22] enables explicit structural modifications, such as changes of the class hierarchy or insertions of new fields and methods. In contrast to AspectJ without annotations, @AspectJ restricts the possibilities of static cross-cutting. For instance, in @AspectJ, it is not possible to insert fields in existing classes.

3.2 Snippets and their Composition

While an aspect in @AspectJ starts with an `@Aspect` annotation, an @J class is annoted with `@J`, which can take some extra annotation parameters.

Snippets are public static methods with void return type annotated with `@BeforeSnippet`, `@AfterSnippet`, `@AfterReturningSnippet`, or `@AfterThrowingSnippet`. These @J annotations correspond to `@Before`, `@After`, `@AfterReturning`, respectively `@AfterThrowing` advice methods in @AspectJ, but may take some additional annotation parameters. In @J, snippets may be woven only before or after a join point; in contrast to @AspectJ, @J does not support weaving around a join point. The @J snippet annotations support the optional boolean parameter `execute` for indicating whether a snippet is to be inlined (default, `execute`=false) or executed at weaving time (`execute`=true).

In contrast to @AspectJ, snippets are always static in @J. Since snippets are inlined or executed at weaving time, it is not possible to change the snippets associated with a program at runtime. In contrast, the standard @AspectJ weaver inserts invocations of advice methods instead of inlining their bodies. The approach taken by @AspectJ has the benefit that the aspect association can be changed at runtime. However, for the purpose of @J, we consider static snippets appropriate, because snippet inlining is a prerequisite for passing data between snippets woven into the same method using local variables.

If multiple snippets match a join point, the @J programmer must specify the precedence of snippets. To this end, the @J snippet annotations support an optional integer parameter `order` (snippets with smaller `order` value come first). Weaving produces an error, if the order of multiple matching snippets is insufficiently specified.

[6] `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6583051`

3.3 Invocation-local Variables

@J supports the notion of *invocation-local variables* [38], in order to allow efficient data passing in local variables between snippets. The term "invocation-local" was chosen to imply that the scope of an invocation-local variable is one invocation of a woven method. Invocation-local variables are accessed through public static fields that have `@InvocationLocal` annotations. Within snippets, invocation-local variables can be read and written as if they were static fields. For each invocation-local variable accessed in a woven method, a local variable is allocated and the bytecodes that access the corresponding static field are simply replaced with bytecodes for loading/storing from/in the local variable.

Each invocation-local variable is initialized in the beginning of a woven method with the value stored in the corresponding static field, which is assigned only during execution of the static initializer.[7] This implies that the @J class holding the snippets and the invocation-local variables is also loaded in the JVM, although the snippets are never invoked at runtime, and the static fields corresponding to invocation-local variables are assigned values only by the corresponding static initializer. As an optimizing, the initialization of local variables corresponding to invocation-local variables in woven methods is skipped, if the weaver can statically determine that the first access is a write.

3.4 Snippet Execution at Weaving Time

In many instrumentation-based software engineering tools, we found optimizations where some static analysis is performed at instrumentation time in order to decide whether and how to instrument a particular location in the bytecode. Standard AspectJ does not support the execution of custom analysis code at weaving time, making it impossible to recast such optimizations in aspects.

@J introduces *executable snippets*, which are executed at weaving time. Executable snippets produce weavable results by writing to invocation-local variables. In the woven method, a bytecode sequence is inserted that reproduces the values of the written invocation-local variables. Executable snippets may access only static context information, such as static join points, and must not have any side effects apart from writing to invocation-local variables of primitive type or of type `java.lang.String`. Executable snippets must not read any invocation-local variable.

Instead of inlining an executable snippet, the @J weaver creates an environment that enables snippet execution for matching join points at weaving time. To this end, the weaver generates a class holding the executable snippets (in a transformed version) and the invocation-local variables. The transformed snippets are instrumented so as to provide the set of written invocation-local variables upon completion. The resulting class is loaded at weaving time, and for each matching join point, the corresponding transformed snippet method is called, passing

[7] The Java memory model [19, 25, 18] ensures that the value assigned by the static initializer is visible to all threads.

the needed static context information of the join point as arguments. Note that this may require allocating static join point instances at weaving time, if an executable snippet makes use of it. After snippet execution, the weaver inlines a bytecode sequence in the woven method that assigns the written invocation-local variables with the respective constants (which are added to the constant pool of the class holding the woven method).

A typical use of an executable snippet is to run some static analysis at weaving time, producing a boolean value in an invocation-local variable indicating whether the join point shall be instrumented. The executable snippet is composed with a normal (inlined) snippet, which is a conditional statement on the value of the boolean invocation-local variable. Evidently, in case the condition is false, the inlined snippet is dead code, which is likely to be eliminated by the just-in-time compiler of the JVM. @J does not perform any bytecode optimization, such as dead code elimination, since we assume that the snippet-based software engineering tools will execute on standard, state-of-the-art JVMs, which already include sophisticated optimizations upon bytecode compilation. A detailed example involving an executable snippet is presented below.

4 Case Study: Recasting LiLa

In this section, we discuss an example @J program that recasts the listener latency profiler LiLa [21].

Listener latency profiling helps developers locate slow operations in interactive applications, where the perceived performance is directly related to the response time of event listeners. LiLa[8] is an implementation of listener latency profiling based on ASM [28], a low-level bytecode engineering library.

The response time for handling an event relates to the execution time of an invoked method on an instance of a class implementing the `java.util.EventListener` interface. In order to reduce profiling overhead, LiLa does not instrument all methods in each subtype of `java.util.EventListener`, but restricts the instrumentation to those methods that are declared in an interface. Hence, LiLa analyzes the class hierarchy to determine which methods to instrument. This optimization reduces profiling overhead at runtime, because less methods are instrumented.

Even though it is possible to recast the basic profiling functionality of LiLa as an aspect in AspectJ, for example, using the `around` advice to measure response time by surrounding the execution of every event-related method, the optimization that reduces the number of instrumented methods cannot be performed at weaving time.

In Figure 1, we show how the static analysis at weaving time is implemented in @J with the executable snippet `analyzeNeedsProfiling(...)`. The result of the snippet execution at weaving time is stored in the invocation-local variable `needsProf`. The `takeStartTime()` snippet, which is inlined after the bytecodes

[8] `http://www.inf.usi.ch/phd/jovic/MilanJovic/Lila/Welcome.html`

that result from executing `analyzeNeedsProfiling(...)`, records the starting time only if the static analysis determined that profiling was needed. The invocation-local variable `start` stores the starting time for later use in the same woven method. The `takeEndTimeAndProfile(...)` snippet intercepts (both normal and abnormal) method completion. The listener object is made accessible within the body of the snippet through the expression "`this(listener)`" in the pointcut declaration. Whenever the execution time exceeds the given threshold, the method `profileEvent(...)` (not shown in the figure) logs an identifier of the intercepted method (conveyed by the static join point), the target object, and the execution time. This information helps developers locate the causes of potential performance problems due to slow event handling.

The example in Figure 2 illustrates the weaving of an `EventListener` implementation. For the sake of easy readability, we show the transformations conceptually at the Java level, whereas the @J weaver operates at the bytecode level. The method `actionPerformed(ActionEvent)` is declared in the implemented interface and needs to be profiled, whereas the method `notDeclaredInInterface()` does not require profiling. Figure 2(b) shows the result of weaving. The interesting part is how the result of the static analysis is stored in the invocation-local variable `needsProf`. The woven code is quite long, since there is a significant amount of dead code. A state-of-the-art compiler will detect and eliminate the dead code, yielding the optimized code shown in Figure 2(c).

5 Related Work

The AspectBench Compiler (*abc*) [5] eases the extension of AspectJ with new pointcuts [1, 11, 14]. Even though the new pointcuts of @J could be implemented as an extension using *abc*, we opted for an annotation-based snippet development style in order to rapidly prototype @J features and therefore focus on the weaving part, rather than on the aspect language front-end. In addition, adapting the @AspectJ weaver to use FERRARI [7] for full method coverage turned out to cause less development effort than modifying *abc*.

Nu [17] enables extensions using an intermediate language model and explicit join points [32]. *Nu* adopts a fine-grained join point. Similar to @J, it allows to express aspect-oriented constructs in a flexible manner. While *Nu* is based on a customized JVM, @J is compatible with standard JVMs and uses standard Java compilers.

Steamloom [10] provides AOP support at the JVM level, which results in efficient runtime weaving. Steamloom enables the dynamic modification and reinstallation of method bytecodes and provides dedicated support for managing aspects. Steamloom uses its own aspect language and provides a parser to support AspectJ-like pointcuts. Steamloom is based on the Jikes RVM [2] and supports thread-locally deployed aspects. In order to support thread safety, Steamloom uses code snippets that are inserted before every call to advices, so as to verify whether the advice invocation for the current thread should be active

```
@J
public class LiLa {
  // listeners executing less than 100 ms (100,000,000 ns) are not logged
  public static final long THRESHOLD_NS = 100L * 1000L * 1000L;

  @InvocationLocal
  public static long start; // stores starting time of listener execution

  @InvocationLocal
  public static boolean needsProf; // stores result of static analysis

  // pointcut matching the execution of any method
  // in any subtype of the EventListener interface
  @Pointcut( "execution(* java.util.EventListener+.*(..))" )
  void listenerExec() {}

  // static analysis at weaving time (result stored in invocation-local variable);
  // jpsp provides method details (package, class, name, signature)
  @BeforeSnippet( pointcut = "listenerExec";
                  execute = true;
                  order = 1; )
  public static void analyzeNeedsProfiling(JoinPoint.StaticPart jpsp) {
    needsProf = isInterfaceMethod(jpsp); // not shown here
  }

  // store starting time upon listener entry, if the static analysis
  // considers profiling necessary
  @BeforeSnippet( pointcut = "listenerExec";
                  order = 2; )
  public static void takeStartTime() {
    if (needsProf) start = System.nanoTime();
  }

  // profile listener execution upon completion, if the static analysis
  // considers profiling necessary and the execution time exceeds the threshold
  @AfterSnippet( pointcut = "listenerExec && this(listener)"; )
  public static void takeEndTimeAndProfile(JoinPoint.StaticPart jpsp,
                                         java.util.EventListener listener) {
    if (needsProf) {
      long exectime = System.nanoTime() - start;
      if (exectime >= THRESHOLD_NS)
        profileEvent(jpsp, listener, exectime); // not shown here
    }
  }
  ...
}
```

Fig. 1. Listener latency profiler LiLa expressed in @J

or not. Similar to Steamloom, PROSE [31] also provides aspect support within the JVM. PROSE combines bytecode instrumentation and aspect support at the just-in-time compiler level with an extension of the Jikes RVM. Unfortunately, these approaches require a customized JVM, thus limiting extensibility and portability.

Prevailing AspectJ weavers do not support the execution of custom analysis code at weaving time, which typically only depends on static information. SCoPE [3] is an AspectJ extension that partially solves this problem by allowing analysis-based conditional pointcuts. Similarly, the approach described in [24]

(a) Before weaving:

```
class ExampleListener implements ActionListener {
  public void actionPerformed(ActionEvent e) { doSomething(); }
  public void notDeclaredInInterface() { doSomethingElse(); }
  ...
}
```

(b) Woven code:

```
class ExampleListener implements ActionListener {
  private static final JoinPoint.StaticPart
    jpsp1 = ..., // representing actionPerformed
    jpsp2 = ...; // representing notDeclaredInInterface

  public void actionPerformed(ActionEvent e) {
    long start = 0L;
    boolean needsProf = true;
    if (needsProf) start = System.nanoTime();
    try { doSomething(); }
    finally {
      if (needsProf) {
        long exectime = System.nanoTime() - start;
        if (exectime >= LiLa.THRESHOLD_NS)
          LiLa.profileEvent(jpsp1, this, exectime);
      }
    }
  }

  public void notDeclaredInInterface() {
    long start = 0L;
    boolean needsProf = false;
    if (needsProf) start = System.nanoTime();
    try { doSomethingElse(); }
    finally {
      if (needsProf) {
        long exectime = System.nanoTime() - start;
        if (exectime >= LiLa.THRESHOLD_NS)
          LiLa.profileEvent(jpsp2, this, exectime);
      }
    }
  }
  ...
}
```

(c) Optimized code (e.g., by a just-in-time compiler that eliminates dead code):

```
class ExampleListener implements ActionListener {
  private static final JoinPoint.StaticPart
    jpsp1 = ..., // representing actionPerformed
    jpsp2 = ...; // representing notDeclaredInInterface

  public void actionPerformed(ActionEvent e) {
    long start = System.nanoTime();
    try { doSomething(); }
    finally {
      long exectime = System.nanoTime() - start;
      if (exectime >= LiLa.THRESHOLD_NS)
        LiLa.profileEvent(jpsp1, this, exectime);
    }
  }

  public void notDeclaredInInterface() { doSomethingElse(); }
  ...
}
```

Fig. 2. Weaving and optimization of an example EventListener implementation

enables customized pointcuts that are partially evaluated at weaving time. @J supports custom analysis through snippets that are executed during the weaving.

Maxine [34] is a meta-circular research VM implemented in Java. Maxine uses a layered compiler with different intermediate representations. Instead of writing the code in a particular intermediate representation to add a runtime feature, Maxine allows developers to write snippets directly in Java, which are compiled into the corresponding intermediate representation. This approach decouples runtime features from compiler work. The Ovm [29] virtual machine follows a similar approach, where a high-level intermediate representation eases the customization for building language runtime systems, so as to define new operations and to modify the semantics of existing ones.

6 Conclusion

Low-level bytecode instrumentation is a prevailing technique for implementing tools that perform some kind of dynamic program analysis, such as profiling. As implementing instrumentations at the bytecode level is tedious and error-prone, specifying dynamic analysis tools with high-level AOP is a promising approach for reducing tool development costs, for improving maintainability, and for easing extension of the tools.

Unfortunately, many prevailing AOP frameworks, such as AspectJ, lack certain features that are important for developing efficient dynamic analysis tools for certain purposes. We identified two missing features in AspectJ that we consider essential for tool development: efficient data passing between woven advices in local variables, and the execution of custom static analyses at weaving time.

In this paper, we propose the annotation-based AOP framework @J, which is based on @AspectJ and incorporates support for these two features. As examples, we recast an existing tool based on low-level bytecode manipulation as an @J program, illustrating the use of @J's distinguishing features. The resulting tool is compactly implemented within a few lines of code.

Regarding limitations, @J suffers from the same problem as any other framework relying on Java bytecode instrumentation. The JVM imposes strict limits on certain parts of a class file (e.g., the method size is limited); these limits may be exceeded by the code inserted upon aspect weaving. Our approach aggravates this issue by inlining snippets, usually increasing the code bloat. Nonetheless, we have not yet encountered any problems due to code growth in practice.

7 Acknowledgements

The work presented in this paper has been supported by the Swiss National Science Foundation as part of the project FERRARI (project number 200021-118016/1).

References

1. S. Akai, S. Chiba, and M. Nishizawa. Region pointcut for AspectJ. In *ACP4IS '09: Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 43–48, New York, NY, USA, 2009. ACM.
2. B. Alpern, C. R. Attanasio, J. J. Barton, B. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, N. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.
3. T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 161–172, New York, NY, USA, 2007. ACM.
4. S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In J. Vitek, editor, *ECOOP '08: Proceedings of the 22th European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 542–565, Paphos, Cyprus, 2008. Springer-Verlag.
5. P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98, New York, NY, USA, 2005. ACM Press.
6. L. D. Benavides, R. Douence, and M. Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 183–202, New York, NY, USA, 2008. Springer-Verlag New York, Inc.
7. W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM Press.
8. W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009. `http://dx.doi.org/10.1002/spe.890`.
9. W. Binder, A. Villazón, D. Ansaloni, and P. Moret. @J - Towards Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine. In *VMIL '09: Proceedings of the 3th Workshop on Virtual Machines and Intermediate Languages for emerging modularization mechanisms*, New York, NY, USA, 2009. ACM.
10. C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 109–124, New York, NY, USA, 2006. ACM.
11. E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, July 20-24 2008*, pages 155–165, New York, NY, USA, 07 2008. ACM.
12. F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 221–230, New York, NY, USA, 2008. ACM.

13. S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.
14. B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 60–71, New York, NY, USA, 2008. ACM.
15. M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.
16. B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.
17. R. Dyer and H. Rajan. Nu: A dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, pages 191–202, New York, NY, USA, 2008. ACM.
18. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
19. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, 2005.
20. E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, New York, NY, USA, 2004. ACM.
21. M. Jovic and M. Hauswirth. Measuring the performance of interactive applications with listener latency profiling. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 137–146, New York, NY, USA, 2008. ACM.
22. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.
23. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
24. K. Klose, K. Ostermann, and M. Leuschel. Partial evaluation of pointcuts. In *PADL*, pages 320–334, 2007.
25. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM.
26. P. Moret, W. Binder, and A. Villazón. CCCP: Complete calling context profiling in virtual execution environments. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 151–160, Savannah, GA, USA, 2009. ACM.
27. NetBeans. The NetBeans Profiler Project. Web pages at `http://profiler.netbeans.org/`.
28. ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.

29. K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 67–76, New York, NY, USA, 2003. ACM.
30. D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.
31. A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, New York, NY, USA, 2003. ACM Press.
32. H. Rajan. A Case for Explicit Join Point Models for Aspect-Oriented Intermediate Languages. In *VMIL '07: Proceedings of the 1st Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms*, page 4, New York, NY, USA, 2007. ACM.
33. D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting Static Source Views in IDEs with Dynamic Metrics. In *ICSM '09: Proceedings of the 2009 IEEE International Conference on Software Maintenance*, pages 253–262, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.
34. Sun Microsystems, Inc. The Maxine Virtual Machine. Web pages at `http://research.sun.com/projects/maxine/`.
35. Sun Microsystems, Inc. JVM Tool Interface (JVMTI) version 1.1. Web pages at `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, 2006.
36. The Apache Jakarta Project. The Byte Code Engineering Library (BCEL). Web pages at `http://jakarta.apache.org/bcel/`.
37. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.
38. A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Advanced Dynamic Runtime Adaptation for Java. In *GPCE '09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*. ACM, Oct. 2009.
39. A. Villazón, W. Binder, and P. Moret. Aspect Weaving in Standard Java Class Libraries. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 159–167, New York, NY, USA, Sept. 2008. ACM.
40. A. Villazón, W. Binder, and P. Moret. Flexible Calling Context Reification for Aspect-Oriented Programming. In *AOSD '09: Proceedings of the 8th International Conference on Aspect-oriented Software Development*, pages 63–74, Charlottesville, Virginia, USA, Mar. 2009. ACM.
41. G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 151–160, New York, NY, USA, 2008. ACM.

Automatic Tool Generation from Structural Processor Descriptions

Florian Brandner
Institute of Computer Languages
Christian Doppler Laboratory - Compilation Techniques for Embedded Processors
Technische Universität Wien

Automatic Tool Generation from Structural Processor Descriptions[1]

The success of embedded systems in mobile communication and entertainment devices, in commodity appliances, the domestic environment, as well as in the (safety) critical control systems of cars and airplanes made these small computer systems an indispensable part of every bodies daily lives. The demands on these systems in terms of reliability, efficiency, and computational power are steadily rising, while at the same time the physical dimensions and costs per unit are expected to shrink for every new product generation.

Application specific instruction processors (ASIPs) have become a valuable tool to deliver high computing power under rigid power and area constraints. The development of such a processor is delicate task that requires intimate knowledge of processor design, software development, compilers, and of course the particular problem domain.

Processor description languages – often referred to as architecture description languages (ADLs) – are a promising approach to capture the behavior, structure, and instruction set of an ASIP using a compact and concise description. These languages typically provide various views of the processor at different abstraction levels: (1) the behavioral level is primarily concerned with the abstract behavior of individual instructions, while (2) the structural view defines the hardware organization and resources. Processor description languages that focus on the former are often called behavioral languages, those following the latter approach structural. If a language combines specifications of both layers, it is called mixed. From a processor model, specified in one of these languages, software development tools, such as a compiler, linker, or assembler, can be (semi-)automatically customized for that particular processor. In addition, simulation tools, test cases, and even hardware models can be derived. Mixed languages typically provide the most flexibility in terms of these applications, because both a rather high-level, but also a rather low-level view of the processor is provided.

In this work a structural processor description language is presented that allows to derive a behavioral model from its structural specifications automatically. The language captures the behavioral and structural details of a processor and thus provides great flexibility, but avoids redundancies known from mixed languages. We demonstrate the feasibility of our approach using two generators: (1) a compiler generator that automatically derives a highly optimizing

[1] This work was supported in part by the Christian Doppler Forschungsgesellschaft and OnDemand Microelectronics.

code generator and (2) a simulator generator that derives a high-speed simulator based on dynamic binary translation. Experiments show that the derived tools can compete with hand crafted tools. The code produced by the generated compilers achieves speedups of up to 20%, on average moderate slowdowns of 5-15% have been observed. The simulation framework is similarly competitive and achieves a simulation speed that often matches the speed of the processor implementation in hardware.

Inline Caching meets Quickening

Stefan Brunthaler
Technische Universität Wien

Inline caches effectively eliminate the overhead implied by dynamic typing. Unfortunately, inline caching is mostly used in code generated by just-in-time compilers. We present efficient implementation techniques for using inline caches without dynamic translation, thus enabling future interpreter implementers to use this important optimization technique—we report speedups of up to 1.4—without the additional implementation and maintenance costs incurred by using a just-in-time compiler.

Typsicheres und generisches MapReduce

Jens Dörre
Universität Passau

MapReduce ist ein bei Google entwickeltes Programmiermodell, das dort die Entwicklung datenparalleler Programme für die massiv verteilte Ausführung ermöglicht. Dieses Modell basiert theoretisch auf lange bekannten Konzepten aus der Funktionalprogrammierung.

Im praktischen Einsatz fehlt allerdings noch ein formales und technisches Fundament, auf dessen Basis Eigenschaften wie Sicherheit und Korrektheit von MapReduce-Programmen garantiert werden können. Auch ist noch unklar, wie dieses Modell generisch in verschiedenen Szenarien angewendet werden kann.

Utilizing Multiple Hardware Threads with Pipeline Parallelism

M. Anton Ertl*, TU Wien

Abstract. In recent years general-purpose CPUs have aquired multiple hardware threads by providing multiple cores per CPU (multi-core) and multiple hardware threads per core (simultaneous multi-threading). Making good use of such resources has been a challenge for several decades that has been successfully attacked for scientific applications, but not to a significant extent for general-purpose applications. The main current paradigm for this programming problem is to have explicit threads that share memory and are synchronized by a variety of synchronzation constructs. Unfortunately, it seems to be too hard to program profitably in this paradigm for general-purpose applications. Pipeline parallelism is a programming paradigm that has proved so easy to understand that shell programmers use it even on machines that have only one hardware thread. In this work we present the case for better support for pipeline parallelism in programming languages, and present ideas on how to improve the scalability of the implementation of pipeline parallelism.

1 Introduction

PCs have gained more than one hardware thread in the last few years. This poses the challenge of utilizing all the threads that the hardware makes available. While multi-processor computers have existed for decades, and there has also been a huge amount of research in parallel computing, this research has focused on scientific applications, the main use of multi-processors in earlier times.

However, PCs are usually not used for scientific applications, and the characteristics of general-purpose applications differ from scientific applications in ways that affect parallelisation; e.g., loop trip counts are typically high for scientific applications, but low for general-purpose applications [Lar91].

2 Background

2.1 Threads and tasks

The hardware supplies thread contexts (e.g., one core on a machine with multiple cores but without simultaneous multi-threading (SMT), or one thread on a core

* Correspondence to: M. Anton Ertl, Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; Email: anton@mips.complang.tuwien.ac.at

that supports SMT). The operating system (OS) supplies OS threads and it allocates hardware threads dynamically to OS threads as it sees fit. Programming languages often also provide something that is often called threads or tasks, and the implementation of the programming language can map these programming language features to OS threads in a 1:1 fashion, or try to avoid system call overhead by using a more complicated scheme.

In this paper, when we discuss the implementation, in particular, the mapping of programming language constructs to OS constructs, we use *threads* to mean OS or hardware threads, and *tasks* to mean programming language tasks.

2.2 How much parallelism is there?

Is there actually parallelism in non-numeric applications? Some of the limits studies that looked at the dynamic data flow dependencies of application runs found high levels of parallelism in non-numeric applications [LW92]. We only have to find out how to organize the data flow into threads (for thread-level parallelism).

2.3 Why is parallel programming hard?

Parallel programming introduces two new classes of errors: When a *race conditions* occurs, the outcome of the program depends on the specific order in which certain actions in different threads are executed. To avoid race conditions, programmers can use synchronization constructs (e.g., locks or semaphores); that can lead to the other class of error: *deadlocks*, where several threads wait for each other. These errors (especially race conditions) are often not deterministically reproducible, which makes it much harder to reveal them in testing than most bugs in sequential programs, and, even after they have shown up, they are still harder to debug.

If the threads access shared memory (a common feature of multi-threaded environments), another problem is that the hardware does not implement an intuitive memory model like sequential consistency. The programmer can use memory barriers to work around that problem, but memory barriers are expensive, so programmers want to avoid them; but again, if they fail to put in a memory barrier that's necessary, the result will be a race condition, but the problem will be even harder to find and fix, because understanding the bug requires understanding the counter-intuitive behaviour of the hardware.

Common synchronization constructs interact in ways that undermine the modularity of the program. So, the main weapon we use to keep programs manageable loses its edge once we decide to parallelize programs using such synchronization constructs.

Parallel programming is also hard is because the goal is to increase the performance. So the goal is not just to have a program that exposes more parallelism, the goal is that the result should run in less real time than the original sequential program. The problems here are that, on one hand, the speedups are limited by

the number of CPUs (so it really only pays off after you have squeezed all the available sequential performance from the program, which does its own damage).

But even worse, parallelizing a program introduces overheads: thread creation and destruction, synchronization, and barriers. So you cannot just parallelize whatever parts look parallelizable, because then the overheads will tend to eat up the potential speedup and more. Instead, parallelization has to be performed looking at the whole program, and finding out where the large parallelization opportunities are while avoiding overheads. Again, this undermines modularization, and it is hard to achieve in a typical general-purpose program where most of the run-time is spent in a number of places, not just one or a few inner loops.

Another problem is that, while some patterns of parallelism (such as independent loop iterations, aka data or DOALL parallelism) are easy to scale across a wide range of thread counts, these patterns do not tend to dominate in general-purpose applications. So the result of parallelizing will often not be very scalable to varying thread counts. The result will be that either the hardware threads will be underutilized, or that there will be more synchronization overhead than necessary.

Overall, the common ways of supporting thread-level parallelism are either too hard to use for widespread use in application programming (conventional synchronization constructs), or they are often not applicable for general-purpose applications (data parallelism).

2.4 Transactions

In recent years there has been a huge amount of research into transactional memory, especially software transactional memory [LR06]. This provides transactions to the programmers, i.e., a sequence of statements that is (logically) executed atomically. This concept is easier to program than traditional synchronization. However, in this work I look in a different direction, for the following reasons:

- There is already lots of research into transactions, and most of the gold in that area probably has already been mined, whereas other directions have received comparatively little attention.
- Transactions are used in a shared-memory setting, which may not be the best approach to modularity.
- Software transactional memory has a significant implementation cost, in complexity and in performance.

3 Pipeline parallelism

Unix pipelines provide a form of parallelism that avoids many of the problems discussed above:

- It is so easy to use pipelines that shell programmers use them even when parallelism is not the goal.

- Unix pipelines are constructed from reusable modules (called filters). Pipeline parallelism is not just compatible with modularisation, it provides an additional form of modularisation.
- Pipelines can be used in general-purpose applications. E.g., pipelines in Unix are often used for text processing and other non-numerical applications. Compilers used to be organized in multiple filter-like passes. General-purpose applications often contain complex control flow (e.g., a recursive graph walk), which makes it hard to divide work for data parallelism, but pipeline stages can contain such complex control flow without problems.

Therefore, we believe that adding pipeline parallelism to general-purpose programming languages would be a good idea, both for parallelism and for modularization. One other benefit that we will focus on in this paper is that one can implement pipeline parallelism in a way that scales from many threads down to few or one.

3.1 Stream languages

Stream languages were designed for implementing digital signal processing (DSP) algoritms on highly parallel hardware [ADK+04,TLM+04]; pipeline (or stream) parallelism is a central feature of these programming languages. Later, such languages were also implemented on conventional CPUs [GR05,ZLRA07]. These implementations were designed for huge data sets and implemented the filters as working on large (256KB and more) blocks, with the following filter only scheduled on a block after the previous filter has finished it, i.e., no direct communication.

3.2 XJava

The general-purpose language XJava [OPT09] is an extension of Java that supports pipeline and data parallelism. The current implementation uses an approach similar to the implementations of stream languages on general-purpose processors, i.e., with filters working on blocks and then returning to a scheduler which then assigns another ready task to the thread.

The advantages of this implementation approach are: It allows a uniform handling of data parallelism and pipeline parallelism. And, if the pipeline has enough data to process, it scales nicely across a wide number of threads (as long as the program provides enough parallelism); short-running and long-running pipeline stages are balanced automatically.

The downside is that there is a ramp-up in parallelism at the start of a pipeline and a ramp-down in the end; with a large block size, these ramps can take a while, and if there are only few blocks, the parallelism will be severely limited. If all the data fits in one block between stages, all the stages are run sequentially and no parallel execution occurs. While large amounts of data may be taken for granted in DSP applications, that is not necessarily the case for general-purpose applications (e.g., compilers).

4 Fine-grained pipeline implementation

This section presents some ideas on how to implement pipelines in programming languages in a way that does not suffer from these problems.

In order to reduce low parallelism from ramp-up and ramp-down effects, we could reduce the block size. However, this increases the overhead, because the scheduler is invoked once for every block.

Instead, we propose using a single-writer single-reader ring buffer fifo for communicating between pipeline stages. This allows communication between the pipeline stages at a fine granularity: cache line granularity would be good for efficiency in the usual case, but even finer granularity is possible. At the same time, large amounts of data can be transferred without involving a scheduler.

Of course, avoiding the scheduler works only if all pipeline stages run at the same time in different threads, and if each pipeline stage produces data at the same rate as the next stage consumes it. Otherwise, some of the ring buffers will sooner or later become full or empty, and either the producer (if full) or the consumer (if empty) will block and won't fully utilize the thread it has available.

How can this scenario be avoided? And what can we do if we don't have or get enough threads to run all the filters at the same time?

There is another efficient and fine-grained implementation of pipelines that does not need multiple threads or frequent scheduler invocations: coroutining. In coroutining one pipeline stage passes control of the thread directly to the next one (on writing to the next stage) or to the previous one (when reading from the previous stage); this just requires saving a few registers of one task and restoring a few registers of the other task (most notably, the instruction and stack pointers).

Now we can reduce the number of threads needed by our pipeline stages by putting a number of consecutive stages into one thread and letting them communicate with each other through coroutining; the communication with stages in other threads is still through ring buffers.

This arrangement does not do anything for balancing the production and consumption rates of the different sub-pipelines. However, this can be achieved by moving pipeline stages between threads when the ring buffer approaches the full or the empty state.

E.g., consider a ring buffer that approaches emptyness: Obviously the writer to this ring buffer is too slow and the reader too fast. This can be remedied by moving the last pipeline stage (B) of the writing thread T1 to the reading thread T2.

In more detail, this works as follows: the writing stage B (still in T1) tells the reader C that B switches to the thread T2 (which contains C), then it switch the output of the previous stage A to use a ring buffer instead of coroutining, and transfers control of its current thread T1 to A. When C reads from the B-C ring buffer, it finds the switching message from B. First it completely consumes all the data in the B-C ring buffer. When that is empty, it switches its reader to use coroutining from B instead of reading from the B-C ring buffer; so when it needs to read after draining the B-C ring buffer, it will do a coroutine call to

B; at some point B will need to read something, and will read it from the A-B ring buffer.

Since T2 now needs more processing time than before and T1 needs less, the ring buffer between A and B should drain slower than the one between B and C used to, or it might even fill up. If it still drains overall, then A could move to T2, too; if the buffer fills up and approaches fullness, then B can be transferred back to T1. By transfering B back and forth between the threads now and then, both threads can be fully employed. This can all happen without any central scheduler being involved.

This means that the overheads of pipelining are relatively small, so an application can be divided into many pipeline stages without incurring much overhead, even if only a few threads are available. And if even less overhead is desired (at the cost of lower flexibility), several pipeline stages could be merged into one fused stage at compile time.

This approach puts certain demands on the operating system scheduler for good performance: all the threads of the application should run at the same time. Otherwise, i.e., if the OS preempts one thread, that thread will be stuck in some state unknown to the other threads, so other threads cannot pick up pipeline stages from that thread. So the other threads will soon fill or drain all the ring buffers involved, and they will eventually block themselves waiting for the descheduled thread. So, if the OS needs to deal with more runnable threads than the hardware provides, our approach will not lead to good performance.

The good news is that in that scenario the hardware is obviously utilized well anyway; conversely, on a typical PC that is mostly idle, that scenario will not happen frequently, and once the load drops, everything will run smoothly again.

Also, with cooperation from the OS, this scenario can be avoided: The OS should either provide gang scheduling, or should have a way of communicating with the application that it needs a thread for other purposes. Then pipeline stages can migrate away from one thread, and the thread can be returned to the OS while the other threads keep running.

5 Conclusion

Increasing the performance of general-purpose programs by parallelizing them is hard for several reasons: Simple data parallelism is often not present due to dependences, or cannot be exploited profitably due to low loop trip counts and short iteration running times. Using the usual synchronization constructs is complex and undermines the modularity. And a balance has to be found between splitting the application into threads and the overheads of thread creation and synchronization, otherwise performance will decrease rather than increase; and finding that balance again undermines the modularity.

Pipeline parallelism can be used for general-purpose programs, and it enhances modularity by splitting programs into reusable filters. Stream languages and XJava implement pipeline parallelism by letting each filter run on a block of data and then calling the scheduler.

We propose an implementation that is based on using coroutining and single-reader single-writer ring buffers. The ring buffers are used for communicating between threads, and coroutining is used for having several pipeline stages in one thread, to reduce the number of threads (for less capable hardware), or to balance the work between different threads. Pipeline stages can move between threads (and the communication switch between coroutining and ring buffer) to dynamically balance the pipeline so that all threads are always employed.

References

[ADK+04] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. Evaluating the imagine stream architecture. In *Proc. 31th Ann. Intl Symp. on Computer Architecture (31th ISCA'04)*, pages 14–25, Munich, Germany, 2004.

[GR05] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *38th Annual International Symposium on Microarchitecture (MICRO-38)*, 2005.

[Lar91] J. R. Larus. Parallelism in numeric and symbolic programs. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Programming, pages 331–349. Pitman, London, 1991.

[LR06] James Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *The* 19th *Annual International Symposium on Computer Architecture (ISCA)*, pages 46–57, 1992.

[OPT09] Frank Otto, Victor Pankratius, and Walter F. Tichy. XJava: Exploiting parallelism with object-oriented stream programming. In *Euro-Par '09*, pages 875–886. Springer LNCS 5704, 2009.

[TLM+04] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matthew Frank, Saman P. Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proc. 31th Ann. Intl Symp. on Computer Architecture (31th ISCA'04)*, pages 2–13, 2004.

[ZLRA07] David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A lightweight streaming layer for multi-core execution. In *2007 Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, 2007.

A Sound, Complete and Usable Hoare-Style Logic for a Sequential Java Subset

Christoph Feller
Technische Universität Kaiserslautern

Proving a Hoare-style logic sound and complete is not a new idea. In [1] we find a formalization of a rather large subset of sequential Java together with a Hoare-style logic and a proof of its soundness and completeness. Why do we do it again?

Our long-term goal is to find a way to modularize reasoning about programs. So we have to find a proper notion of modules but also a way to reason about these. An axiomatic semantic seems to be the best way to do the latter. So this work can be seen as a preparation for our long-term goals: To acquire more insight into program logics and a better understanding of the used theorem prover Isabelle/HOL.

Another aim of this work is to provide a more usable logic than [1]. That essentially means to keep the important definition and especially the rules of the logic as legible and concise as possible. As a technical detail we will show how the locale mechanism of Isabelle/HOL contributed towards this aim.

References

[1] David Oheimb, Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic

Reinventing Haskell Backtracking

Sebastian Fischer

Institut für Informatik
Christian-Albrechts Universität zu Kiel

Backtracking is an approach to explore different alternatives during search. It is often implemented in imperative programming languages by storing an explicit stack of choice points. In functional programs backtracking can be expressed by using lazy lists of results and list concatenation to express alternatives. A well-known more efficient alternative to lazy lists uses a sophisticated combination of so-called success- and failure continuations [1]. Our work sheds new light on this implementation and recasts it in order to make it more intuitively accessible.

Instead of directly coming up with success- and failure continuations, we develop them in two steps. First, we employ the concept of difference lists to overcome the inefficiency of list concatenation and then we use continuations to efficiently combine search operations sequentially. Both techniques are established functional programming folklore but they have never been applied in combination to reformulate continuation based backtracking. In fact, we were pleasantly surprised to see that he combination of difference lists with continuations leads to a well-known implementation of backtracking rather than a new one.

Our exploration of this insight lead to new implementations of other search strategies that are simpler than previous implementations in Haskell. By using specific different types instead of difference lists, our approach leads to new implementations of breadth-first- and iterative-deepening depth-first search. We obtain breadth-first search by representing levels of the search space in distinct lists, and depth-bound search by representing the search space by a function that takes a depth limit. The additional plumbing with continuations allows to separate these simple ideas from the problem of combining different searches sequentially which significantly simplifies both implementations.

The details of our approach as well as experiments to evaluate it are described elsewhere [2] and not reproduced here, due to copyright restrictions.

References

1. Hinze, R.: Deriving backtracking monad transformers. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2000) 186–197
2. Fischer, S.: Reinventing Haskell Backtracking. In: INFORMATIK 2009, Im Focus das Leben, 39. Jahrestagung der Gesellschaft für Informatik. LNI, GI (2009)

Concurrency Engineering with S-Net

Clemens Grelck[1,2], Sven-Bodo Scholz[2], and Alex Shafarenko[2]

[1] University of Amsterdam, Institute of Informatics
Science Park 107, 1098 XG Amsterdam, Netherlands
c.grelck@uva.nl
[2] University of Hertfordshire, School of Computer Science
Hatfield, Herts, AL10 9AB, United Kingdom
{c.grelck,s.scholz,a.shafarenko}@herts.ac.uk

Abstract. We present the design of S-Net, a coordination language and component technology based on stream processing. S-Net boxes integrate existing sequential code as stream-processing components into highly asynchronous concurrent streaming networks. Their construction is based on algebraic formulae built out of four network combinators. S-Net achieves a near-complete separation of concerns between application code, written in a conventional programming language, and coordination code, written in S-Net itself. Subtyping on the level of boxes and networks and a tailor-made inheritance mechanism achieve flexible software reuse.

1 Introduction

The recent advent of multicore technology in processor designs [1] has introduced parallel computing power to the desktop. Unlike the increase in clock frequency characteristic of previous generations of processors, application programs do not automatically benefit from multiple cores, but require explicit parallelisation. This need brings parallel and distributed programming techniques from the niche of traditional supercomputing application areas into the main stream of software engineering. This shift demands new programming concepts, tools and infrastructure that are suitable for average application programmers not previously exposed to the pitfalls of concurrent program execution.

Parallel programming in the conventional style is considered notoriously difficult because it intertwines two different aspects od program execution: algorithmic behaviour, i.e. what is to be computed, and organisation of concurrent execution, i.e. how a computation is performed on multiple execution units including the necessary problem decomposition, communication and synchronisation. S-Net [2] is a novel declarative coordination language whose design thoroughly avoids the intertwining of computational and organisational aspects through active separation of concerns: S-Net completely separates the concern of writing sequential application building blocks (i.e. *application engineering)* from the the concern of composing these building blocks to form a parallel application (i.e.*concurrency engineering).*

More precisely, S-NET defines the coordination behaviour of networks of asynchronous, stateless components and their orderly interconnection via typed streams. We deliberately restrict S-NET to coordination aspects and leave the specification of the concrete operational behaviour of basic components, named *boxes* in S-NET terminology, to conventional languages. An S-NET box is connected to the outside world by two typed streams, a single input stream and a single output stream. Data on these streams is organised as non-recursive records, i.e. collections of label-value pairs. The operational behaviour of a box is characterised by a stream transformer function that maps a single record from the input stream to a (possibly empty) stream of records on the output stream. In order to facilitate dynamic reconfiguration of networks, a box has no internal state and any access to external state (e.g. file system, environment variables, etc.) is confined to using the streaming network. Boxes execute fully asynchronously: as soon as a record is available on the input stream, a box may start computing and producing records on the output stream.

The restriction to a single input stream and a single output stream per box again again is motivated by separation of concurrency engineering from application engineering. If a box had multiple input streams, this would immediately raise the question as to what extent input data arriving on the various input streams is synchronised. Do we wait for exactly one data package on each input stream before we start computing like in Petri nets? Or do we alternatively start computing when the first data item arrives and see how far we get without the other data? Or could we even consume varying numbers of data packages from the various input streams? This immediately intertwines the question of synchronisation, which is a classical concurrency engineering concern, with the concept of the box, which in fact is and should only be an abstraction of a sequential compute component.

The same is true for the output stream of a box. Had a box multiple output streams, this would immediately raise the question of data routing, again a classical concurrency engineering concern, as the box code would need to decide to which stream data should be sent. Having a single output stream oinly, in contrast, clearly seperates the routing aspect from the computing aspect of the box and, thus, concurrency engineering from application engineering.

The construction of streaming networks based on instances of asynchronous components is a distinctive feature of S-NET: Thanks to the restriction to a single-input/single-output stream component interface we can describe entire networks through algebraic formulae. Network combinators either take one or two operand components and construct a network that again has a single input stream and a single output stream. As such a network again is a component, construction of streaming networks becomes an inductive process. We have identified a total of four network combinators that prove sufficient to construct a large number of network prototypes: static serial and parallel composition of heterogeneous components as well as dynamic serial and parallel replication of homogeneous components.

Structural subtyping on records greatly facilitates adaptation of individual components to varying contexts. More precisely, components only need to be specific about record fields that are actually needed for the associated computation or that are (at least potentially) created by that computation. In excess to these required fields, however, an input record to some component may have an arbitrary number of further fields. These additional fields bypass the component and are added to any outgoing record through an automatic coercion mechanism, named *flow inheritance*.

To summarise, the motivation of S-Net is to completely separate algorithmic programming from concurrency engineering. Indeed any user-defined box represents an algorithm encapsulated in the form of a function that performs a computation on a data item (its argument list) and which computes and passes back to the environment one or more similar data items. Any communication or synchronisation actions, any division of work between workers and gathering of the data back in one place is happening in the coordination language. This makes boxes unit-testable in isolation, and also removes (due to the requirement of statelessness) any placement or mobility constraints from all user-defined boxes. This is in sharp contrast with SPMD programming styles inherent in MPI and similar parallel libraries, and to the best of our knowledge, any other coordination language.

The remainder of the paper is organised as follows. In Section 2 we sketch out the S-Net type system. Sections 3 and 4 introduce boxes and networks, respectively. We demonstrate their interaction by a small programming example in Section 5 and conclude in Section 6.

2 The type system of S-Net

2.1 Record types

The type system of S-Net is based on non-recursive variant records with *record subtyping*. Informally, a *type* in S-Net is a non-empty set of anonymous *record variants* separated by vertical bars. Each record variant is a possibly empty set of named *record entries*, enclosed in curly brackets. We distinguish two different kinds of record entries: *fields* and *tags*. A field is characterised by its *field name* (label); it is associated with an opaque value at runtime. Hence, fields can only be generated, inspected or manipulated by using an appropriate box language. A tag is represented by a name enclosed in angular brackets. At runtime tags are associated with integer values, which are visible to both box language code and S-Net. The rationale of tags lies in controlling the flow of records through a network. They should not be misused to hold box language data that can be represented as integer values.

We illustrate S-Net types by a simple example from 2-dimensional geometry: For instance, we may represent a rectangle by the S-Net type

```
{x, y, dx, dy}
```

providing fields for the coordinates of a reference point (x and y) and edge lengths in both dimensions (dx and dy). Likewise, we may represent a circle by the center point coordinates and its radius:

```
{x, y, radius}
```

Using the S-Net support for variant record types we may easily define a type for geometric bodies in general, encompassing both rectangles and circles:

```
{x, y, dx, dy} | {x, y, radius}
```

Often it is convenient to name variants. In S-Net this can be done using tags:

```
{<rectangle>, x, y, dx, dy} | {<circle>, x, y, radius}
```

S-Net supports type definitions; we refer the interested reader to [2] for details.

2.2 Record subtyping

S-Net supports structural subtyping on record types. Subtyping essentially is based on the subset relationship between sets of record entries. Informally, a type is a subtype of another type if it has additional record entries in the variants or additional variants. For example, the type

```
{<circle>, x, y, radius, colour}
```

representing coloured circles is a subtype of the previously defined type

```
{<circle>, x, y, radius}  .
```

Likewise, we may add another type to represent triangles:

```
  {<rectangle>, x, y, dx, dy}
| {<circle>, x, y, radius}
| {<triangle>, x, y, dx1, dy1, dx2, dy2};
```

which again is a supertype of

```
  {<rectangle>, x, y, dx, dy}
| {<circle>, x, y, radius}
```

as well as a supertype of

```
{<circle>, x, y, radius, colour}  .
```

Our definition of record subtyping coincides with the intuitive understanding that a subtype is more specific than its supertype(s) while a supertype is more general than its subtype(s). In the first example, the subtype contains additional information concerning the geometric body (i.e. its colour) that allows us to distinguish, for instance, green circles from blue circles. In contrast, the more general supertype identifies all circles regardless of their colour. In our second example, the supertype is again more general than its subtype as it encompasses all three different geometric bodies. Subtype {<circle>,x,y,radius,colour} is more specific than its supertypes because it rules out triangles and rectangles from the set of geometric bodies covered. Unlike subtyping in object-oriented languages our definition of record subtyping is purely structural; {} (i.e. the empty record) denotes the most common supertype.

2.3 Type signatures

Type signatures describe the stream-to-stream transformation performed by a box or a network. Syntactically, a type signature is a non-empty set of type mappings each relating an *input type* to an *output type*. The input type specifies the records a box or network accepts for processing; the output type characterises the records that the box or network may produce in response. For example, the type signature

```
{a,b} | {c,d} -> {<x>} | {<y>} , {e} -> {z}
```

describes a network that accepts records that either contain fields `a` and `b` or fields `c` and `d` or field `e`. In response to a record of the latter type the network produces records containing the field `z`. In all other cases, it produces records that either contain tag `x` or tag `y`.

2.4 Flow inheritance

Up-coercion of records upon entry to a certain box or network creates a subtle problem in the stream-processing context of S-Net. In an object-oriented setting the control flow eventually returns from a method invocation that causes an up-coercion. While during the execution of the specific method the object is treated as being one of the respective superclass, it always retains its former state in the calling context. In a stream-processing network, however, records enter a box or network through its input stream and leave it through its output stream, which are both connected to different parts of the whole network. If an up-coercion results in a loss of record entries, this loss is not temporary but permanent.

The permanent loss of record entries is neither useful nor desirable. For example, we may have a box that manipulates the position of a geometric body regardless of whether it is a rectangle, a circle or a triangle. The associated type signature of such a box could be as simple as {`x,y`}->{`x,y`}. This box would accept circles, rectangles and triangles focusing on their common data (i.e. the position) and ignoring their individual specific fields and tags. Obviously, we must not lose this data as a consequence of the automatic up-coercion of complete geometric bodies to type {`x,y`}. Hence, we complement this up-coercion with an automatic down-coercion. More precisely, any field or tag of an incoming record that is not explicitly named in the input type of a box or network bypasses the box or network and is added to any outgoing record created in response, unless that record already contains a field or tag with the same label. We call this coercion mechanism *flow inheritance*.

As an example, let us assume a record {`<circle>,x,y,radius`} hits a box {`x,y`}->{`x,y`}. While fields `x` and `y` are processed by the box code, tag `circle` and field `radius` bypass the box without inspection. As they are not mentioned in the output type of the box, they are both added to any outgoing record, which consequently forms a complete specification of a circle again.

3 Box abstractions

3.1 User-defined boxes

From the perspective of S-NET boxes are the atomic building blocks of streaming networks. Boxes are declared in S-NET code using the key word `box` followed by a box name as unique identifier and a box signature enclosed in round brackets. The box signature very much resembles a type signature with two exceptions: we use round brackets instead of curly brackets, and we have exactly one type mapping that has a single-variant input type. For example,

```
box foo ((a,b,<t>) -> (a,b) | (<t>));
```

declares a box named `foo`, which accepts records containing (at least) fields `a` and `b` plus a tag `t` and in response produces records that either contain fields `a` and `b` or tag `t`. Boxes are implemented using a box language rather than S-NET. It is entirely up to the box implementation to decide how many output records a box actually emits and of which of the output variants they are. This may well depend on the values of the input record entries and, hence, can only be determined at runtime.

```
snet_handle_t *foo( snet_handle_t *handle,
                    int *a, mytype_t *b, int t)
{
    /* some computation on a, b and t */
  snetout( handle, 1, a, b);
    /* some computation */
  snetout( handle, 2, t);
  return( handle);
}
```

Fig. 1. Example box function implementation in C

Box signatures use round brackets instead of curly brackets to express the fact that in box signatures sequence does matter. (Remember that type signatures are true sets of mappings between true sets of record entries.) Sequence is essential to support a mapping to function parameters of some box language implementation rather than using inefficient means such as string matching of field and tag names. For example, we may want to associate the above box declaration `foo` with a C language implementation in the form of the C function `foo` shown in Fig. 1.

The entries of the input record type are effectively mapped to the function parameters in their order of appearance in the box signature. We implement record fields as opaque pointers to some data structure and tags as integer values. In addition to the box-specific parameters the box function implementation always receives an opaque S-NET handle, which provides access to S-NET internal data.

Since boxes in S-NET generally produce a variable number of output records in response to a single input record, we cannot exploit the function's return

value to determine the output record. Instead, we provide a special function `snetout` that allows us to produce output records during the execution of the box function, as demonstrated in Fig. 1. The first argument to `snetout` is the internal handle that establishes the necessary link to the execution environment. The second argument to `snetout` is a number that determines the output type variant used. So, the first call to `snetout` in the above example refers to the first output type variant. Consequently, the following arguments are two pointers. The second call to `snetout` refers to the second output type variant and, hence, a single integer value follows. Eventually, the box function returns the handle to signal completion to the S-NET context.

This is just a raw sketch of the box language interfacing. Concrete interface implementations may look differently to accommodate characteristics of certain box languages, and even the same box language may actually feature several interface implementations with varying properties.

3.2 Filter boxes

The filter box in S-NET is devoted to housekeeping operations. Effectively, any operation that does not require knowledge of field values can be expressed by this versatile built-in box in a simpler way than using an atomic box and a fully-fledged box language implementation. Among these operations are

- elimination of fields and tags from records,
- copying fields and tags,
- adding tags,
- splitting records,
- simple computations on tag values.

Syntactically, a filter box is enclosed in square brackets and consists of a type (pattern) to the left of an arrow symbol and a semicolon-separated sequence of filter actions to the right of the arrow symbol, for example:

```
[{a,b,<t>} -> {a} ; {c=b,<u=42>} ; {b,<t=t+1>}]
```

This filter box accepts records that contain fields `a` and `b` as well as tag `t`. In general, the type-like notation to the left of the arrow symbol acts as a pattern on records; any incoming record's type must be a subtype of the pattern type.

As a response to each incoming record, the filter box produces three records on its output stream. The specifications of these three records are separated by semicolons to the right of the arrow symbol. Outgoing records are defined in terms of the identifiers used in the pattern. In the example, the first output record only contains the field `a` adopted from the incoming record (plus all flow-inherited record entries). The second output record contains field `b` from the input record, which is renamed to `c`. In addition there is a tag `u` set to the integer value 42. The last of the three records produced contains the field `b` and the tag `t` from the input record, where the value associated with tag `t` is incremented by one. S-NET supports a simple expression language on tag values that essentially consists of arithmetic, relational and logical operators as well as a conditional expression.

3.3 Synchrocells

The synchrocell is the only "stateful" box in S-Net. It also provides the only means in S-Net to combine two existing records into a single one, whereas the opposite direction, the splitting of a single record, can easily be achieved by both user-defined boxes and built-in filter boxes. Syntactically, a synchrocell consists of an at least two-element comma-separated list of type patterns enclosed in `[|` and `|]` brackets, for example

```
[| {a,b,<t>}, {c,d,<u>} |]
```

The principle idea behind the synchrocell is that it keeps incoming records which match one of the patterns until all patterns have been matched. Only then the records are merged into a single one that is released to the output stream. Matching here means that the type of the record is a subtype of the type pattern. The pattern also acts as an input type specification: a synchrocell only accepts records that match at least one of the patterns.

A synchrocell has storage for exactly one record of each pattern. When a record arrives at a fresh synchrocell, it is kept in this storage and is associated with each pattern that it matches. Any record arriving thereafter is only kept in the synchrocell if it matches a previously unmatched pattern. Otherwise, it is immediately sent to the output stream. As soon as a record arrives that matches the last remaining previously unmatched variant, all stored records are released. The output record is created by merging the fields of all stored records into the last matching record. If an incoming record matches all patterns of a fresh synchrocell right away, it is immediately passed to the output stream.

Although we called synchrocells "stateful" above, this is only true as far as individual records are concerned. Synchrocells nevertheless realise a functional mapping from input stream to output stream as a whole.

4 Streaming networks

4.1 Network definitions

User-defined and built-in boxes form the atomic building blocks for stream processing networks; their hierarchical definition is at the core of S-Net. As a simple example of a network definition take:

```
net X {
  box foo ((a,b)->(c,d));
  box bar ((c)->(e));
}
connect foo..bar;
```

Following the key word `net` we have the network name, in this case `X`, and an optional block of local definitions enclosed in curly brackets. This block may contain nested network definitions and box declarations. Hierarchical network definitions incur nested scopes, but in the absence of relatively free variables the scoping rules are straightforward.

A distinctive feature of S-Net is the fact that complex network topologies are not defined by some form of wire list, but by an expression language. Each network definition contains such a topology expression following the key word `connect`. Atomic expressions are made up of box and network names defined in the current scope as well as of built-in filter boxes and synchrocells. Complex expressions are inductively defined using a set of network combinators that represent the four essential construction principles in S-Net: serial and parallel composition of two (different) networks as well as serial and parallel replication of one network, as sketched out in Fig. 2. Note that any network composition again yields a network with exactly one input and one output stream.

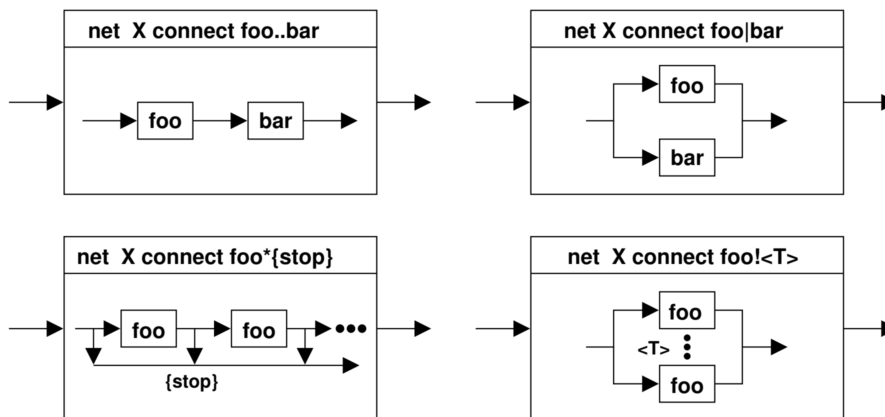


Fig. 2. Illustration of network combinators and their operational behaviour: serial composition (top-left), parallel composition (top-right), serial replication (bottom-left) and indexed parallel replication (bottom-right)

4.2 Serial composition

The binary serial combinator "`..`" connects the output stream of the left operand to the input stream of the right operand. The input stream of the left operand and the output stream of the right operand become those of the combined network. The serial combinator establishes computational pipelines, where records are processed through a sequence of computational steps.

In the example of Fig. 2, the two boxes `foo` and `bar` are combined into such a pipeline: all output from `foo` goes to `bar`. This example nicely demonstrates the power of flow inheritance: In fact the output type of box `foo` is not identical to the input type of box `bar`. By means of flow inheritance, any field `d` originating from box `foo` is stripped off the record before it goes into box `bar`, and any record emitted by box `bar` will have this field be added to field `e`.

In contrast to box declarations, type signatures of networks are generally inferred by the compiler. For example the inferred type signature of the network `X` in the above example is `{a,b}->{d,e}`. Type inference is a particularly interesting aspect of S-NET. We refer the interested reader to [3] for a thorough treatment of the subject.

4.3 Parallel composition

The binary parallel combinator "`|`" combines its operands in parallel. Any incoming record is sent to exactly one operand depending on its own type and the operand type signatures. The output streams of the operand networks (or boxes) are merged into a single stream, which becomes the output stream of the combined network. Fig. 2 illustrates the parallel composition of two networks `foo` and `bar` (i.e. `foo|bar`).

To be precise, any incoming record is sent to that operand network whose type signature's input type is matched best by the record's type. Let us assume the type signature of `foo` is {`a`}`->`{`b`} and that of `bar` is {`a,c`}`->`{`b,d`}. An incoming record {`a,<t>`} would go to box `foo` because it does not match the input type of box `bar`, but thanks to record subtyping does match the input type of box `foo`. In contrast, an incoming record {`a,b,c`} would go to box `bar`. Although it actually matches both input types, the input type of box `bar` scores higher (2 matches) than the input type of box `foo` (1 match). If a record's type matches both operand type signatures equally well, the record is non-deterministically sent to one of the operand networks.

4.4 Serial replication

The serial replication combinator "`*`" replicates the operand network (the left operand) infinitely many times and connects the replicas by serial composition. The right operand of the combinator is a type (pattern) that specifies a termination condition. Any record whose type is a subtype of the termination type pattern (i.e. matches the pattern) is released to the combined network's output stream.

In fact, an incoming record that matches the termination pattern right away is immediately passed to the output stream without being processed by the operand network at all. This coincidence with the meaning of star in regular expressions particularly motivates our choice of the star symbol. Fig. 2 illustrates the operational behaviour of the star combinator for a network `foo*`{`<stop>`}: Records travel through serially combined replicas of `foo` until they match a given type pattern, more precisely the type of the record is a record subtype of the specified type (pattern). Optionally, the exit pattern may be refined by a boolean expression on the values of the tags in the type pattern. Actual replication of the operand network is demand-driven. Hence, networks in S-NET are not static, but generally evolve dynamically, though in a restricted way.

4.5 Indexed parallel replication

Last but not least, the parallel replication combinator "!" takes a network or box as its left operand and a tag as its right operand. Like the star combinator, it replicates the operand, but connects the replicas using parallel rather than serial composition. The number of replicas is conceptually infinite. Each replica is identified by an integer index. Any incoming record goes to the replica identified by the value associated with the given tag. Hence, all records that have the same tag value will be routed to the same replica of the operand network. Fig. 2 illustrates the operational behaviour of indexed serial replication for a network `foo!<T>`. In analogy to serial replication, instantiation of replicas is demand-driven.

Note that this construct in combination with serial replication allows dynamic, SPMD style connections: a network such as `(A!<P>)*<Y>` allows the box `A` to receive records with a certain value of `<P>` and create records with either the same or different value of `<P>` which will be fed to an appropriate replica of `A`. Any output from `A` that is meant to be released should be tagged with `<Y>`. It is quite obvious that dynamic communication could be made as complex as the programmer requires using more combinators, but crucially the only routing issue that is dealt with dynamically is *which* replica of a box a given record should be directed to, not which box, and since all replicas share the same type signature, S-Net remains type safe even under dynamic routing.

4.6 Putting it all together

The restriction of every box and every network to a single input stream and a single output stream allows us to describe complex streaming networks in a very concise way using algebraic formulae rather than wire lists. Fig. 3 demonstrates the power of our approach by means of an example network

```
net XYZ connect ((A..B|C..D)!<i>)*{<stop>}  .
```

The example uses 4 predefined boxes: `A`, `B`, `C` and `D`. Sequential compositions of `A/B` and `C/D`, respectively, are combined in parallel. The resulting subnetwork is replicated vertically through indexed parallel replication and, thereafter, horizontally through serial replication. Although Fig. 3 demonstrates the complexity of this network, its specification takes no more than half a line of code.

To conclude this section, we wish to make a remark on the implementation. Space limitations do not permit us to touch on any details; however, the crucial point of S-Net implementation is the use of nondeterminism. Solutions for parallel computing tend to be deterministic since nondeterminism can affect the values being processed and can lead to incorrect results. However, in the context of stream processing nondeterminism manifests itself as the lack of order in a stream sequence. Obviously if the receiver box either does not need to receive the records in a certain order, or if the required order can be reconstructed at the output of the top-level network from the stream content, nondeterministic merges are safe. On the other hand, the use of nondeterministic merges dramatically reduces the latency of processing, since the implementation is free to

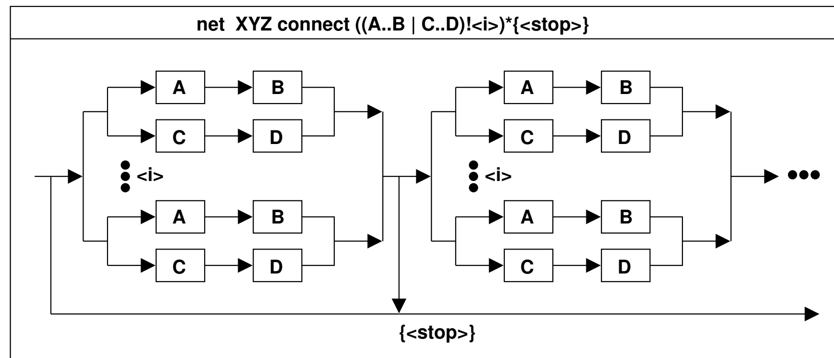


Fig. 3. Example of complex network construction with S-NET network combinators

merge streams in the order of arrival rather than queuing off records that have overtaken ones created earlier. Also in a multistage pipelined processing scheme, a record can represent work to be done by several algorithms in any order. For example, by a box A that determines the maximum row elements of a matrix and divides the rows by them, and by a box B that exchanges the left and right halves of all rows. A solution such as `X..((A..B)|(B..A))` where A and B present the same input type will be of this kind. The nondeterministic merger of the parallel combinator will be in a position to use its nondeterminism to choose the pathway depending on how busy boxes `A` and `B` are.

5 Example: solving Sudoku puzzles

We illustrate the potential of S-NET by a simple search problem: finding solutions to sudoku puzzles. While sudokus are simple enough to be explored in detail, they are computationally non-trivial as they require search over an imbalanced tree of theoretically up to 9^{81} possibilities. In this sense, sudokus act as an interesting, albeit simple, model of a real-world search problem.

Sudokus are played on a 9 by 9 board of numbers. Starting out from a board with several given numbers, the overall aim is to fill all empty positions with numbers so that the following conditions hold: (i) each row contains the numbers 1 to 9 exactly once, (ii) each column contains the numbers 1 to 9 exactly once, and (iii) each of the nine 3 by 3 sub-boards contains the numbers 1 to 9 exactly once. Although in general we may have an arbitrary number of solutions or no solution at all, well constructed sudokus have a unique solution.

Fig. 4 shows the S-NET implementation of a simple Sudoku solver; a textual specification of the same solver can be found in Fig. 5. The first box (`computeOpts`) expects records that only contain an abstract representation of a Sudoku board, i.e. the search problem to be solved. The box computes all potential settings for each open position in the Sudoku board (`opts`).

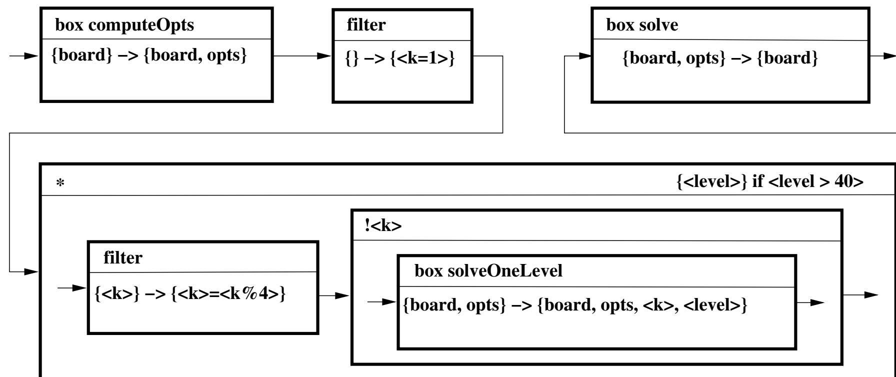


Fig. 4. Solving Sudoku puzzles with S-Net: graphical illustration

The solver itself is embedded within a serial replication: The box `solveOneLevel` tries to fix one further position on the board. For each possible number at that position it outputs a record containing the new board, the new (further) options and a tag `<level>`, whose value signals the number of fixed positions. In principle, the unfolding of the search tree could continue until all solutions are found, but on many target architectures it is more useful to control the amount of unfolding. In the example, we achieve this by refining the termination condition of the serial replication: as soon as 41 of the 81 positions of the board have been fixed, a record leaves the serial replication and is solved by the subsequent box `solve` without further unfolding of the S-Net network.

```
net sudoku {
  box computeOpts ((board)->(board,opts));
  box solveOneLevel ((board,opts)->(board,opts,<k>,<level>));
  box solve ((board,opts)->(board));
  net oneLevel connect [{<k>}->{<k=k%4>}] .. solveOneLevel!<k>;
}
connect computeOpts
        .. [{}->{<k=1>}]
        .. oneLevel * {<level>} if <level>40>
        .. solve;
```

Fig. 5. Solving Sudoku puzzles with S-Net: textual specification

The additional tag `<k>` in conjunction with indexed parallel replication creates another dimension of parallelism: Within each serial unfolding stage four concurrent instantiations of the `solveOneLevel` box process boards independently. A more thorough presentation of the Sudoku solver including box implementations in the functional array language SaC [4] can be found in [5].

6 Conclusions and future work

We have presented the design of S-Net, a declarative language for describing streaming networks of asynchronous components. Several features distinguish S-Net from existing stream processing approaches.

- S-Net boxes are fully asynchronous components communicating over buffered streams.
- S-Net thoroughly separates coordination aspects from computation aspects.
- The restriction to SISO components allows us to describe complex streaming networks by algebraic formulae rather than error-prone wiring lists.
- We utilise a type system to guarantee basic integrity of streaming networks.
- Data items are routed through networks in a type-directed way making the concrete network topology a type system issue.
- Record subtyping and flow inheritance make S-Net components adaptive to their environment.

The overall design of S-Net is geared towards facilitating the composition of components developed in isolation. The box language interface in particular allows existing code to be turned into an S-Net stream processing component with very little effort.

We have by now completed a prototype implementation of S-Net. This consists of a compiler for S-Net [6], including type inference [3], and box language interfaces for C and SaC [4] that allow us to write complete applications [5]. Furthermore, we have implemented a runtime system based on Posix threads for truly concurrent execution of S-Net programs on general-purpose shared memory multiprocessor and multicore architectures [7] as well as an MPI-based distributed memory runtime system for clusters of such machines [8].

Besides several smaller demonstrator applications we are currently working on a non-trivial plasma physics simulation as well as on a radar-based moving target identification (MTI) application that uses space-time adaptive processing (STAP) to demonstrate the suitability of S-Net to coordinate concurrent activities on a representative scale. In the future we aim at compiling S-Net to novel many-core processor designs like the MicroGrid architecture [9, 10] and to investigate into dynamic reconfiguration and self-adaptivity of S-Net networks [11].

References

1. Sutter, H.: The free lunch is over: A fundamental turn towards concurrency in software. Dr. Dobb's Journal **30** (2005)
2. Grelck, C., Shafarenko, A. (eds):, Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S.B., Shafarenko, A.: S-Net Language Report 1.0. Technical Report 487, University of Hertfordshire, School of Computer Science, Hatfield, England, United Kingdom (2009)
3. Cai, H., Eisenbach, S., Grelck, C., Penczek, F., Scholz, S.B., Shafarenko, A.: S-Net Type System and Operational Semantics. In: Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland. (2008)

4. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multi-threaded execution. International Journal of Parallel Programming **34** (2006) 383–427
5. Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA, IEEE Computer Society Press, Los Alamitos, California, USA (2007)
6. Grelck, C., Penczek, F.: Implementing S-Net: A Typed Stream Processing Language, Part I: Compilation, Code Generation and Deployment. Technical report, University of Hertfordshire, Department of Computer Science, Compiler Technology and Computer Architecture Group, Hatfield, England, United Kingdom (2007)
7. Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In Scholz, S., Chitil, O., eds.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom. Lecture Notes in Computer Science, Springer-Verlag (2009) to appear.
8. Grelck, C., Julku, J., Penczek, F.: Distributed S-Net. In Morazan, M., ed.: Implementation and Application of Functional Languages, 21st International Symposium, IFL'09, South Orange, NJ, USA, Seton Hall University (2009)
9. Bernard, T., Bousias, K., de Geus, B., Lankamp, M., Zhang, L., Pimentel, A., Knijnenburg, P., Jesshope, C.: A Microthreaded Architecture and its Compiler. In Arenez, M., Doallo, R., Fraguela, B., Tourino, J., eds.: Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS'06), Frankfurt/Main, Germany. (2006) 326–342
10. Bousias, K., Jesshope, C., Thiyagalingam, J., Scholz, S.B., Shafarenko, A.: Graph Walker: Implementing S-Net on the Self-adaptive Virtual Processor. In: Proceedings of the Æther-Morpheus Workshop From Reconfigurable to Self-Adaptive Computing (AMWAS'08), Lugano, Switzerland. (2008)
11. Penczek, F., Scholz, S.B., Grelck, C.: Towards Reconfiguration and Self-Adaptivity in S-Net. In Scholz, S.B., ed.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, Hertfordshire, UK. Technical Report 474, University of Hertfordshire, UK (2008) 330–339

Automatische Paketisierung

Sebastian Hack
Universität des Saarlandes, Saarbrücken

„Packetization"(also: Data Parallelization or SIMDfication) is the process of transforming scalar code, given by a CFG G, into a CFG G' that works on N scalar input values at once (where N is the target-architecture's SIMD width). One execution of G' is equivalent to N parallel executing instances of G. This technique is important for data-parallel algorithms, in particular applications in computer graphics such as ray tracing.

The benefit of packetization lies in the usage of SIMD instructions. To this end, a value in the scalar instance is mapped to a SIMD "packet" in the packetized version. However, each of the scalar instances may take different control-flow paths. To avoid splitting SIMD packets apart, control flow is almost completely replaced by predicated execution (only loop-back branches have to be kept).

We present a packetization algorithm and discuss its limitations. Furthermore, we present first results of using our algorithm in the shading system of a ray tracer: the programmer writes scalar code in C/C++ that is then automatically packetized. Although still lacking some optimizations, the packetized CFGs already outperform their scalar counterparts by an average factor of 3.6 using a vector width of 4.

Set Functions for Functional Logic Programming*

Michael Hanus

Christian-Albrechts-Universit"at zu Kiel

Abstract. We propose a novel approach to encapsulate non-deterministic computations in functional logic programs. Our approach is based on set functions that return the set of all the results of a corresponding ordinary operation. A characteristic feature of our approach is the complete separation between a usually-non-deterministic operation and its possibly-non-deterministic arguments. This separation leads to the first provably order-independent approach to computing the set of values of non-deterministic expressions. Our approach solves easily and naturally problems mishandled by current implementations of functional logic languages.

* Joint work with Sergio Antoy, Portland State University. The full paper has been published in the Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2009), ACM Press, 2009, pages 73–82

Verbesserung der Modellbildung und Analyse verteilter Geschäftsprozesse durch Prozessumstrukturierung

Thomas S. Heinze[1], Wolfram Amme[1], Simon Moser[2]

[1] Friedrich-Schiller-Universität Jena
{T.Heinze,Wolfram.Amme}@uni-jena.de
[2] IBM Entwicklungslabor Böblingen
smoser@de.ibm.com

1 Einführung und Problemstellung

In den letzten Jahren konnte eine Entwicklung hin zu dienstorientierten und verteilten Software-Architekturen beobachtet werden. Diese Entwicklung findet auch innerhalb der informationstechnischen Modellierung und Realisierung von Geschäftsprozessen ihren Niederschlag. So lassen sich verteilte Geschäftsprozesse aus bestehenden, zumeist örtlich und organisatorisch unabhängigen Diensten zusammensetzen und derart neue Dienste erzeugen, die ihrerseits Grundlage für weitere Geschäftsprozesse bilden können. Die Web Services Business Process Execution Language (WS-BPEL) [1] ist die wohl derzeit vielversprechenste Spezifikationssprache für verteilte Geschäftsprozesse auf Grundlage der Web-Service-Technologie. Ein verteilter Geschäftsprozess der Sprache WS-BPEL wird als Web Service durch Komposition anderer Web Services, dass heißt durch Festlegen von deren Interaktion, implementiert. Die Interaktion der Web Services erfolgt dabei mittels Nachrichten, die über wohldefinierte Schnittstellen ausgetauscht werden. Eine wichtige Voraussetzung für eine fehlerfreie Komposition ist daher die Kompatibilität der an der Interaktion beteiligten Web Services.

Die Frage nach der Kompatibilität umfasst nicht nur die syntaktische Kompatibilität von Web Services, dass heißt die Übereinstimmung der Schnittstellen. Dies ist insbesondere der Fall, falls die Interaktion der Web Services ein für verteilte Geschäftsprozesse charakteristisches zustandsbehaftetes Kommunikationsprotokoll realisiert. Dann muss auch sichergestellt werden, dass es während der Interkation zu keinem unerwünschten Verhalten, beispielsweise zu Verklemmungen, kommt. In diesem Zusammenhang wird von der Verhaltenskompatibilität gesprochen. So werden zwei Web Services verhaltenskompatibel genannt, falls sie in einer möglichen Umgebung, in der sie miteinander interagieren, ordnungsgemäß terminieren können. Um Fehler während der Laufzeit von verteilten Geschäftsprozessen zu vermeiden, sollte die Verhaltenskompatibilität bereits beim Entwurf der Prozesse überprüft werden. Eine geeignete Methode zur Analyse von Eigenschaften wie der Verhaltenskompatibilität beruht auf der Modellbildung durch Petrinetze [4]. Herkömmliche Abbildungsvorschriften ignorieren jedoch aus Gründen der Analysierbarkeit des Petrinetzmodells die Datenaspekte

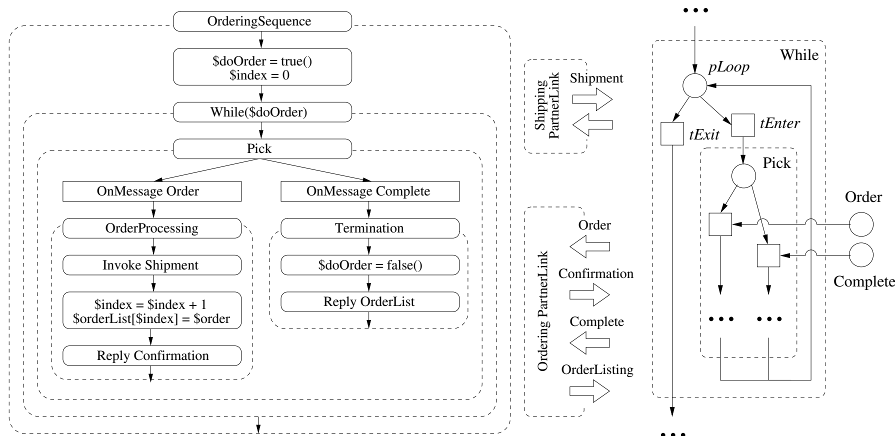


Abb. 1. `OrderingSequence` (links) und Teil des zugehörigen Petrinetzmodells (rechts)

und -abhängigkeiten der Prozesse. Als Folge dieser Abstraktion wird der bedingte Kontrollfluss durch nichtdeterministische Strukturen repräsentiert und derart fehlerhafte Analyseergebnisse, auch für einfache Prozesse, in Kauf genommen.

Ein Prozessfragment das nicht fehlerfrei analysiert werden kann, falls von Datenabhängigkeiten abstrahiert wird, ist in Abbildung 1 angegeben. Die dargestellte Aktivität `OrderingSequence` ist möglicherweise Bestandteil eines größeren Geschäftsprozesses zur Realisierung eines Online-Shops. Darin kann ein Kunde durch wiederholtes Senden der Nachricht `Order` mehrere Waren bestellen, und derart die Auftragsabwicklung für jede einzelne Bestellung einleiten. Nach Übermittlung aller Bestellungen kann er den Bestellvorgang schließlich durch Senden der Nachricht `Complete` abschließen. Zur Umsetzung dieses Protokolls enthält die Aktivität `OrderingSequence` eine Schleife, deren Ausführung durch die boolesche Variable `doOrder` geregelt wird: Ist der Wert der Variablen *true*, wird die Schleife ausgeführt, andernfalls nicht. Anfangs wird der Wert auf *true* gesetzt und so die Schleife ausgeführt. Die `Pick`-Aktivität innerhalb der Schleife aktiviert dann die Sequenz `OrderProcessing`, falls die Nachricht `Order` empfangen wurde. Wurde hingegen `Complete` übermittelt, wird stattdessen der Wert der Variablen `doOrder` auf *false* gesetzt und die Schleife in der Folge beendet.

Wird nun ein passendes Gegenstück zu diesem Prozessfragment betrachtet, dass lediglich eine Bestellung ausführt, so besteht die Interaktion aus den Nachrichten `Order`, `Confirmation`, `Complete` und `OrderList`. Offenbar sind die beiden Prozessfragmente dann verhaltenskompatibel, da es zu keiner Verklemmung kommen kann. Die in der Arbeit [4] beschriebene, petrinetzbasierte Kompatibilitätsanalyse kommt aber zum gegenteiligen Schluss: Die Schleife in `OrderingSequence` wird innerhalb des Petrinetzmodells auf die in Konflikt stehenden Transitionen *tEnter* und *tExit*, zur Modellierung von Schleifenein- und -austritt, abgebildet. Da der dadurch eingeführte Konflikt willkürlich zu lösen ist, kann die Schleife beliebig oft durchlaufen werden. Demnach können Situatio-

nen auftreten, in denen die Aktivität `OrderingSequence` weitere Bestellungen erwartet, obwohl die Nachricht `Complete` bereits empfangen wurde. Es kommt zu einer Verklemmung und die Kompatibilitätsanalyse infolgedessen zu dem Ergebnis, dass die zwei Prozessfragmente nicht kompatibel sind.

Zusammenfassend ist die Kompatibilitätsanalyse in [4] fehleranfällig. Das Weglassen der Datenabhängigkeiten von Schleifen- und Verzweigungsbedingungen führt zu einer zu starken Abstraktion innerhalb der zugrundeliegenden Petrinetzmodellierung. Um die Zahl solcher Analysefehler zu reduzieren, schlagen wir in [2] eine der eigentlichen Kompatibilitätsanalyse vorgelagerte Umstrukturierung von WS-BPEL-Prozessen vor. Diese Umstrukturierung soll im Folgenden anhand des eingeführten Prozessfragments `OrderingSequence` vorgestellt werden. Durch Anwendung unserer Umstrukturierungstechnik lassen sich bedingte Schleifen und Verzweigungen immer dann so transformieren, dass deren Daten- in Kontrollabhängigkeiten umgewandelt werden können, falls die zugehörigen Schleifen- oder Verzweigungsbedingungen zur Laufzeit nur auf Konstanten beliebigen Typs zugreifen. Als Resultat der Transformation entstehen semantisch äquivalente Prozesse, in denen die Schleifen- oder Verzweigungsbedingungen entfernt werden können. Auf diese Weise lässt sich die Anzahl nichtdeterministischer Strukturen innerhalb des Petrinetzmodells verringern und diese mögliche Ursache von fehlerhaften Ergebnissen der Kompatibilitätsanalyse einschränken.

2 Prozessrepräsentation

Zur präzisen Analyse von Geschäftsprozessen der Sprache WS-BPEL wird ein Repräsentationsformat benötigt, das in der Lage ist, sowohl den Kontrollfluss als auch die Datenaspekte und -abhängigkeiten der Prozesse wiederzugeben. Zu diesem Zweck wird eine Erweiterung von Workflow-Graphen [6], einem im Rahmen der Analyse von Geschäftsprozessen zum Einsatz kommenden Repräsentationsformat, genutzt. Da Workflow-Graphen nur den (unbedingten) Kontrollfluss der Prozesse modellieren, müssen sie für eine verlustfreie Prozessrepräsentation erweitert werden. Aufgrund ihrer Ähnlichkeit zu Kontrollflussgraphen können sie aber leicht mit der Concurrent Static Single Assignment Form (CSSA-Form) [3], einem aus dem Übersetzerbau stammenden Format, angereichert werden. Die auf diese Weise erweiterten Workflow-Graphen unterstützen damit auch die Analyse der Datenaspekte und -abhängigkeiten der repräsentierten Prozesse.

Der erweiterte Workflow-Graph für das Prozessfragment `OrderingSequence` ist in Abbildung 2 dargestellt. Aktivitäten sind darin durch Knoten wiedergegeben und Kanten verbinden diese entsprechend dem Kontrollfluss. Die elementaren Aktivitäten des Prozesses, wie beispielsweise `Reply Confirmation`, werden unter Verwendung einzelner Knoten abgebildet. Sequenzen von elementaren Aktivitäten sind dann durch mehrere sukzessive miteinander verbundende Knoten modelliert. Zur Repräsentation der strukturierten Aktivitäten `While` und `Pick` werden zusätzliche Knoten eingefügt, um die Aufspaltung (*Split*, *Pick*) und Vereinigung (*Header*, *Merge*) des Kontrollflusses abbilden zu können.

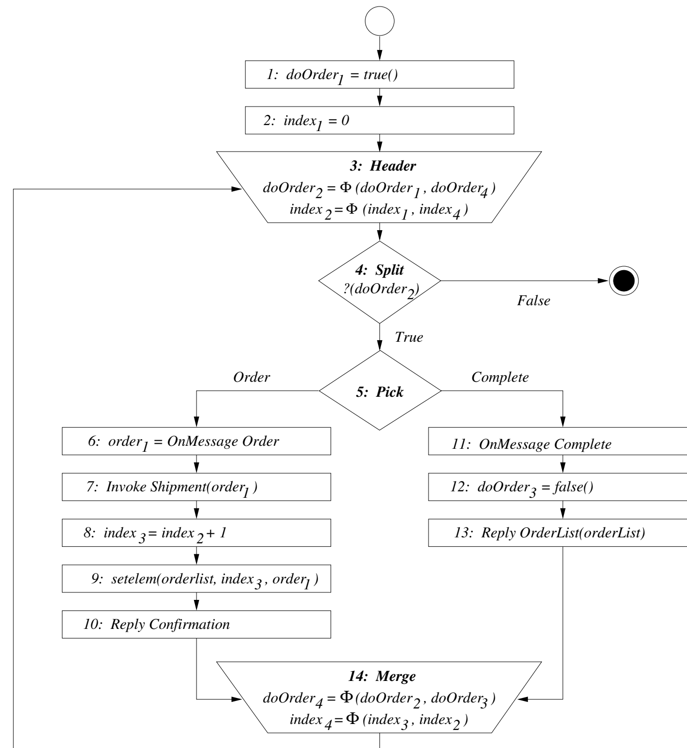


Abb. 2. Erweiterter Workflow-Graph

Wie bereits erwähnt, dient die CSSA-Form in erweiterten Workflow-Graphen der Wiedergabe von Prozessdaten und deren Manipulation. Anwendung und Vorteile dieses Repräsentationsformats für die Analyse von WS-BPEL-Prozessen wurden bereits in der Arbeit [5] beschrieben. Grundlegende Eigenschaft der CSSA-Form bildet die, statisch gesehen, einmalige Definition von Variablen. Zu diesem Zweck werden die Variablen eines Prozesses so umbenannt, dass jede Definition einen eigenen Namen erhält (beispielsweise $doOrder_1, \ldots, doOrder_4$ für Variable `doOrder` in `OrderingSequence`). Dadurch verhalten sich Variablen wie konstante Werte, insbesondere sind die Beziehungen zwischen Definition und Gebrauch einer Variablen explizit repräsentiert. Eine Ausnahme sind Schleifen, da aufgrund der statischen Betrachtungsweise Variablendefinitionen innerhalb von Schleifen als einmalige Definition gelten. Ein besonderes Vorgehen ist zudem notwendig, falls mehrere Definitionen einer Variablen auf verschiedenen Kontrollflusspfaden in einem Knoten zusammentreffen. In diesen Fällen werden Φ-Funktionen eingefügt, um die konkurrierenden Definitionen zusammenzufassen (vergleiche $doOrder_4 = \Phi(doOrder_2, doOrder_3)$ in *Merge*). Die Operanden der Φ-Funktionen bilden dabei gerade die Variablendefinitionen und der Funktionswert entspricht der Definition, die zur Laufzeit tatsächlich ausgeführt wurde.

3 Umstrukturierung

Aufbauend auf dieser Prozessrepräsentation lassen sich die Bedingungen von Schleifen und Verzweigungen analysieren. Die Bedingung der im Prozessfragment `OrderingSequence` enthaltenen Schleife entspricht genau der Variablen $doOrder_2$. Deren Wert wird durch eine Φ-Funktion im Knoten *Header* definiert. Diese führt die konkurrierenden Definitionen der Variablen vor Ausführung der Schleife ($doOrder_1$) und nach Ausführung eines Schleifendurchlaufs ($doOrder_4$) zusammen. Dabei ist die Definition nach Ausführung eines Durchlaufs ebenfalls durch eine Φ-Funktion angegeben, die die Werte auf den zwei möglichen Pfaden innerhalb der Schleife ($doOrder_2$, $doOrder_3$) zusammenfasst. Da alle diese Definitionen entweder einer Konstantenzuweisung oder einer Φ-Funktion entsprechen, hängt der Wert von $doOrder_2$ offenbar lediglich vom Pfad des Kontrollflusses während der Laufzeit ab: Wird die Schleife zum ersten Mal ausgeführt, wird $doOrder_2$ der Wert von $doOrder_1$, also *true*, zugewiesen und die Schleifenbedingung damit erfüllt. Dasselbe gilt für jeden weiteren Durchlauf der Schleife, solange bis die Zuweisung in Knoten *12* ausgeführt wird. Danach wird $doOrder_2$ der Wert von $doOrder_3$, also *false*, zugewiesen. In der Folge ist die Bedingung nicht mehr erfüllt und die Schleifenausführung wird abgebrochen.

Wir nennen Schleifenbedingungen dieser Art, in denen alle Variablen ausschließlich durch ineinander verschachtelte Φ-Funktionen und Konstantenzuweisungen definiert sind, statisch quasi-konstant. Da der Wert einer solchen Bedingung nur vom zur Laufzeit ausgeführten Kontrollflusspfad abhängig ist, können die Datenabhängigkeiten der Bedingung auch durch Kontrollabhängigkeiten repräsentiert werden. Zu diesem Zweck führen wir eine Art Schleifenabrollen durch, indem wir die Schleife durch mehrere Kopien des Schleifenrumpfs ersetzen. Jede Kopie, Schleifeninstanz genannt, entspricht dabei der Ausführung der Schleife für eine bestimmte Belegung der in der Schleifenbedingung vorkommenden Variablen. Auf diese Weise kann die Schleifenbedingung statisch ausgewertet und innerhalb des umstrukturierten Prozessfragments entfernt werden.

Vor Durchführung der eigentlichen Transformation einer Schleife mit statisch quasi-konstanter Schleifenbedingung wird diese in eine Normalform überführt. Die Normalform ist durch die Auftrennung aller Pfade des Kontrollflusses charakterisiert, auf denen in einem beliebigen Schleifendurchlauf verschiedene Werte für die Variablen der Schleifenbedingung definiert werden. Diese Überführung ist für die Schleife des Prozessfragments `OrderingSequence` in Abbildung 3 dargestellt. Die Variable der dort vorhandenen Bedingung ($doOrder_2$) kann, wie oben beschrieben, für einen beliebigen Schleifendurchlauf drei verschiedene Werte annehmen. Die Wahl des Wertes wird durch den zuvor ausgeführten Kontrollflusspfad festgelegt. Um nun die Pfade aufzutrennen, muss der Knoten *Merge* aufgelöst werden, da er zwei der drei möglichen Werte zusammenfasst ($doOrder_2$ und $doOrder_3$). Da der unmittelbare Nachfolger dieses Knotens jedoch gerade dem Kopf der Schleife *Header* entspricht, reicht es dazu aus, die Vorgängerknoten *10* und *13* direkt mit dem Kopf zu verbinden und die Operanden der darin enthaltenen Φ-Funktionen anzupassen (Operand $doOrder_4$ durch $doOrder_2$ und $doOrder_3$ ersetzen). Im Anschluss kann der Knoten *Merge* entfernt werden.

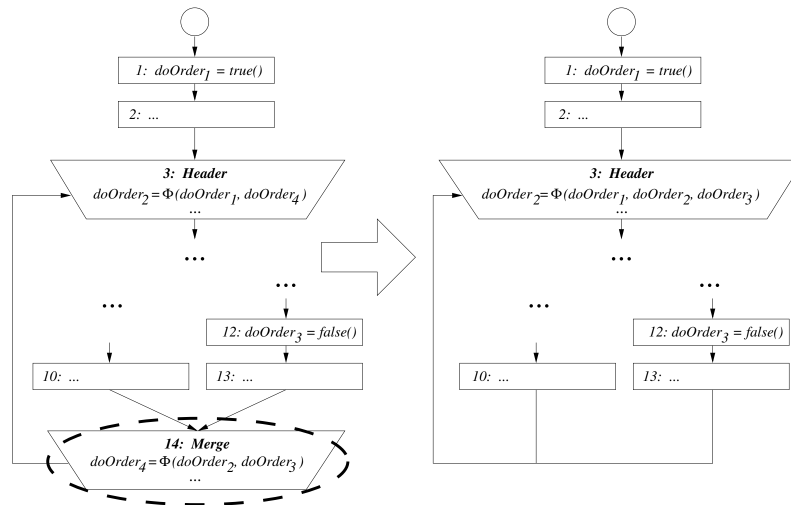


Abb. 3. Überführung der Schleife in Normalform

Im nachfolgenden Transformationsschritt werden dann die Datenabhängigkeiten der Schleife in Kontrollabhängigkeiten umgewandelt, indem die Schleife durch mehrere miteinander verbundene Schleifeninstanzen ersetzt wird. Zur Ableitung der Schleifeninstanzen kann die Normalform der Schleife als eine Art Blaupause benutzt werden. Zu diesem Zweck werden die Φ-Funktionen im Kopf der Schleife aufgelöst und die Überprüfung der Schleifenbedingung vor den Schleifenkopf positioniert. Abbildung 4 zeigt das auf diese Weise abgeleitete Muster für die Schleifeninstanzen des Fragments `OrderingSequence`. Wie zu sehen ist, befinden sich im Kopf der Schleife (*Header*) keine Φ-Funktionen mehr. Stattdessen wurde für jede eingehende Kante ein neuer Knoten (*15*, *16*, *17*) eingefügt, der Zuweisungen der der Kante zugeordneten Operanden an die ursprünglich im Schleifenkopf mittels Φ-Funktionen definierten Variablen enthält. Gleichzeitig wurden als Nachfolger dieser Knoten Kopien des Knotens *Split* mit der Schleifenbedingung eingefügt, die damit nun vor dem Kopf der Schleife überprüft wird (Knoten *18*, *19*, *20*). Da diese Kopien kontrollieren, ob die Schleife betreten wird oder nicht, werden sie im Folgenden auch als Wächterknoten bezeichnet.

Wie bereits erwähnt entspricht eine Schleifeninstanz der Ausführung der Schleife für eine bestimmte Belegung der in der Schleifenbedingung vorkommenden Variablen mit Werten. Dementsprechend lässt sich für eine Belegung dieser Variablen $(var_1, \dots, var_n)$ mit Werten $(v_1, \dots, v_n)$ die zugehörige Instanz erzeugen, indem jede Variable var_i innerhalb des Instanzmusters durch deren Belegung v_i ersetzt wird. Offensichtlich können die Bedingungen in den Wächtern einer Instanz statisch ausgewertet werden, da alle darin vorkommenden Variablen durch Konstanten definiert sind. Während der Transformation wird diese Eigenschaft ausgenutzt, indem die Wächterknoten sukzessive besucht und ausgewertet werden. Ist die Bedingung eines Wächters erfüllt, wird dieser durch eine

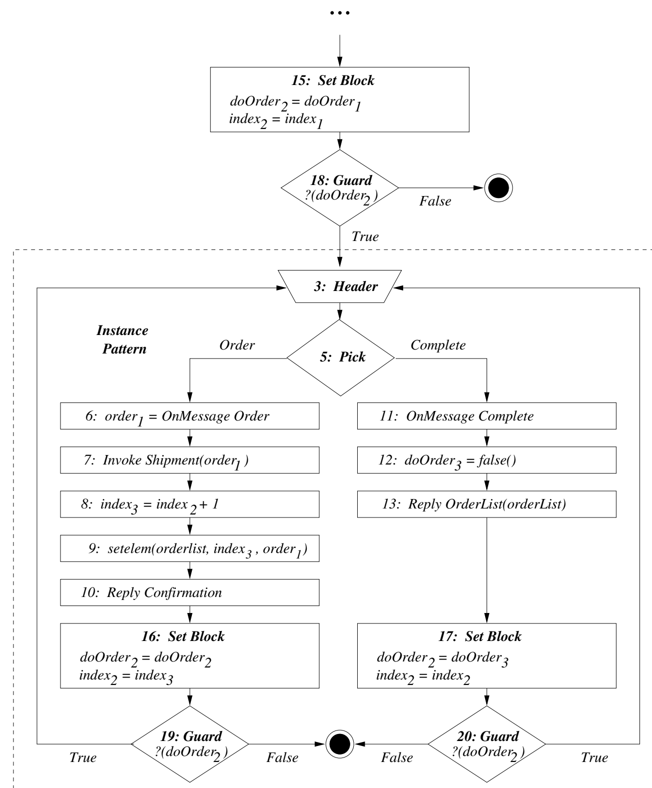


Abb. 4. Muster für Schleifeninstanzen

Kante zu einer Schleifeninstanz ersetzt, ansonsten durch eine Kante zum Austrittsknoten der Schleife. Im ersten Fall richtet sich die Auswahl der jeweiligen Instanz nach der im Vorgängerknoten des Wächters definierten Variablenbelegung. Ist noch keine Schleifeninstanz für eine Belegung vorhanden, wird eine entsprechende neue Instanz aus dem Instanzmuster generiert.

Das Ergebnis der Anwendung dieser Umstrukturierung auf das Prozessfragment `OrderingSequence` ist in Abbildung 5 angegeben. Die Transformation erfolgte in drei Schritten. Im ersten Schritt wurde der Wächter am Schleifeneintritt (Knoten *18*) ausgewertet und durch Erzeugen und Einfügen einer neuen Schleifeninstanz für die Belegung $doOrder_2 = true$ ersetzt. Diese Instanz enthielt wiederum selbst zwei Wächter, mit denen in den folgenden zwei Schritten fortgefahren wurde. Die Auswertung des Wächterknotens *19* ergab ebenfalls *true*. In der Folge wurde dieser Wächter durch eine Kante zu der im vorherigen Schritt bereits erzeugten Instanz $Instance_{doOrder_2=true}$ ersetzt. Da die Auswertung der Bedingung im Wächter *20* hingegen *false* ergab, wurde dieser durch eine Kante zum Austrittsknoten der Schleife ersetzt. Im Anschluss daran waren keine weiteren Wächterknoten mehr vorhanden, so das nur eine Instanz erzeugt wurde.

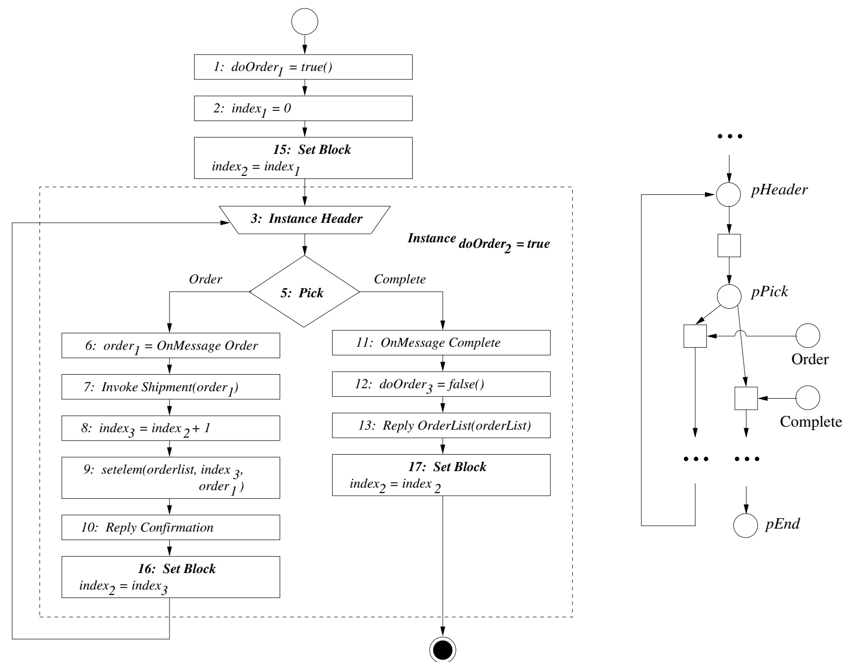


Abb. 5. Umstrukturierter Workflow-Graph (links) und Petrinetzmodell (rechts)

Da die Kompatibilitätsanalyse nach [4] ein petrinetzbasiertes Modell der zu analysierenden Prozesse nutzt, muss der erweiterte Workflow-Graph nach erfolgter Umstrukturierung wieder auf ein Petrinetz abgebildet werden. Zu diesem Zweck lässt sich die in [7] beschriebene, für reguläre Workflow-Graphen definierte Petrinetzsemantik anpassen und ausnutzen. Ein Ausschnitt des resultierenden Petrinetzes ist in Abbildung 5 dargestellt. Wie erwartet, ist darin der nichtdeterministische Konflikt des ursprünglichen Modells nicht mehr vorhanden, da die Schleifenbedingung im umstrukturierten Workflow-Graphen entfernt werden konnte. Dadurch kommt die Kompatibilitätsanalyse des Fragments `OrderingSequence` und des eingangs beschriebenen Gegenstücks nun zum korrekten Ergebnis, dass beide Aktivitäten verhaltenskompatibel sind.

4 Zusammenfassung und Ausblick

Der vorliegende Beitrag beschreibt die in [2] eingeführte Umstrukturierungstechnik für verteilte Geschäftsprozesse der Sprache WS-BPEL. Diese Technik ist in der Lage die Datenabhängigkeiten einer bestimmten Art von Schleifen und Verzweigungen in semantisch äquivalente Kontrollabhängigkeiten zu transformieren. Auf diese Weise können die Prozessmodelle bestehender Analysen präzisiert und so die Verfälschung von Analyseergebnissen eingeschränkt werden, wie sich am Beispiel einer petrinetzbasierten Kompatibilitätsanalyse zeigen lässt.

In darauf aufbauenden Arbeiten soll untersucht werden, inwiefern sich der Ansatz der Umstrukturierung auch zur Auflösung anderer Ausprägungen des bedingten Kontrollflusses anwenden lässt, neben den beschriebenen statisch quasikonstanten Bedingungen. Insbesondere sind wir an Verzweigungen und Schleifen interessiert, für die sich die möglichen Werte der in den Bedingungen vorkommenden Variablen nur teilweise oder überhaupt nicht einschränken lassen. Ein möglichen Ansatzpunkt dazu bieten abstraktere Notationen von Schleifeninstanzen. So sind Instanzen denkbar, in denen die Belegungen von Variablen mit Hilfe von Intervallen oder prädikatenlogischen Ausdrücken beschrieben werden.

Literatur

[1] Alves, Alexandre ; Arkin, Assaf ; Askary, Sid ; Barreto, Charlton ; Bloch, Ben ; Curbera, Francisco ; Ford, Mark ; Goland, Yaron ; Guízar, Alejandro ; Kartha, Neelakantan ; Liu, Canyang K. ; Khalaf, Rania ; König, Dieter ; Marin, Mike ; Mehta, Vinkesh ; Thatte, Satish ; van Rijn, Danny ; Yendluri, Prasad ; Yiu, Alex: *Web Services Business Process Execution Language Version 2.0*. Organization for the Advancement of Structured Information Standards, 2007

[2] Heinze, Thomas S. ; Amme, Wolfram ; Moser, Simon: A Restructuring Method for WS-BPEL Business Processes Based on Extended Workflow Graphs. In: Dayal, Umeshwar (Hrsg.) ; Eder, Johann (Hrsg.) ; Koehler, Jana (Hrsg.) ; Reijers, Hajo A. (Hrsg.): *Proceedings of the 7th International Conference on Business Process Management (BPM 2009), September 8-10, 2009, Ulm, Germany*, Springer-Verlag, 2009 (Lecture Notes in Computer Science 5701), S. 211–228

[3] Lee, Jaejin ; Midkiff, Samuel P. ; Padua, David A.: Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In: Li, Zhiyuan (Hrsg.) ; Yew, Pen-Chung (Hrsg.) ; Huang, Chua-Huang (Hrsg.) ; Chatterjee, Siddharta (Hrsg.) ; Sadayappan, P. (Hrsg.) ; Sehr, David (Hrsg.): *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing (LCPC '97), August 7-9, 1997, Minneapolis, Minnesota, USA*, Springer-Verlag, 1998 (Lecture Notes in Computer Science 1366), S. 114–130

[4] Martens, Axel ; Moser, Simon ; Gerhardt, Achim ; Funk, Karoline: Analyzing Compatibility of BPEL Processes. In: *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006), February 19-25, 2006, Guadeloupe, French Caribbean*, IEEE Computer Society Press, 2006, S. 147

[5] Moser, Simon ; Martens, Axel ; Görlach, Katharina ; Amme, Wolfram ; Godlinski, Artur: Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis. In: *Proceedings of the 2007 IEEE International Conference on Services Computing (SCC 2007), July 9-13, 2007, Salt Lake City, Utah, USA*, IEEE Computer Society Press, 2007, S. 98–105

[6] Sadiq, Wasim ; Orlowska, Maria E.: Analyzing Process Models Using Graph Reduction Techniques. In: *Information Systems* 25 (2000), Nr. 2, S. 117–134

[7] van der Aalst, W. M. P. ; Hirnschall, A. ; Verbeek, H. M. W.: An Alternative Way to Analyze Workflow Graphs. In: Pidduck, A. B. (Hrsg.) ; Woo, C. (Hrsg.) ; Mylopoulos, J. (Hrsg.) ; Ozsu, M. T. (Hrsg.): *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE 2002), May 27-31, 2002, Toronto, Canada*, Springer-Verlag, 2002 (Lecture Notes in Computer Science 2348), S. 535–552

Generation of Incremental Parsers for Modern IDEs

Christoph Höger choeger@cs.tu-berlin.de

Technische Universität Berlin

Abstract. Tools in modern IDEs (*Integrated Developments Environment*) usually work on models derived from the abstract syntax tree. Because those models should be validated and reconciled in real time, parsing the source code is subject to strict limitations in runtime and memory. In this paper we provide a Java based approach to extend a special LR Parser with methods for incremental re-parsing a given text. We also show that while incremental parsers cannot be faster than batch-parsers in general (that is: for any given grammar), our implementation reaches $\mathcal{O}(log(n))$ in the median. We will conclude by showing some optimizations on the approach and measuring their influence on the runtime of the parser.

1 Introduction

This paper will start with a general overview or the parsergenerator that was extended to generate incremental parsers and a motivation for incremental parsing in general. Afterwards we will present the general algorithm, its implementation and some optimization. Finally we will give an example on how the incremental parser works on an example grammar.

1.1 Prerequisites and Conventions

In this paper we will talk about a common parsing technique called *shift-reduce* parsers. We assume the reader is familiar with those kind of parsers and formal languages in general.
As a notational convention when it comes to grammars, Terminals will be written underlined, words as small latin characters and Nonterminals big latin characters. A parser configuration is written as a tuple

$$([S_1, \ldots, S_n\$], [\underline{t_1}, \ldots, \underline{t_k}\$], [T_1, \ldots, T_n\$])$$

with S being the states on the state stack, $\underline{t}$ the terminals on the input stack (that may be written as words) and T the tokens on the token stack. The current stacks top is marked by a \$. Otherwise we use common mathematical notations.

1.2 Related Work

The automated creation of various parsers and their classification in terms of formal languages can be found in [3]. General informations about parsing and compiler techniques are described in depth in [2]. An optimal algorithm for incremental LR parsers is shown in [6].

1.3 PaGe

PaGe (**Pa**rser **Ge**nerator) is an implementation of an experimental LR parser generator developed at the TU-Berlin [5]. PaGe differs from classic LR parsers in two ways: First of all the state machine is not built by using the well known Sets of Items (as seen in e.g. [3]), but by transforming the grammar to directly represent the language of the viable prefixes. Secondly there are two kinds of special symbols added to the grammar: Action symbols and Pseudo Terminals. Both are used to keep track of reductions: An Action Symbol represents a distinct reduction to a Nonterminal while a Pseudo Terminal represents the productions that follow after a reduction had been applied.
After adding Action Symbols to every production in the grammar it is transformed into the so called Normalform where every production has the either the form: $N \rightarrow \alpha Z_i$ or $Z_j \rightarrow \textcircled{n}$ where α is a Terminal or Nonterminal from the original grammar and $\textcircled{n}$ is the Action Symbol that was introduced for the reduction of the Nonterminal N. Note, that this transformation does not change the language of the grammar, because of the way the Z symbols are introduced: A rule $N \rightarrow \alpha\beta\textcircled{n}$ would become $N \rightarrow \alpha Z_1$ with the two rules $Z_1 \rightarrow \beta Z_2$ and $Z_2 \rightarrow \textcircled{n}$ added to the grammar.
In a next step the grammar is 'unfolded': Every original Nonterminal is replaced

$$
\begin{aligned}
E &\rightarrow V\underline{\pm}V[A]\textcircled{e} \\
A &\rightarrow \underline{(}E\underline{)}\textcircled{a} \\
V &\rightarrow X \\
&\ \ \mid\ Y \\
X &\rightarrow \underline{x}\textcircled{x} \\
Y &\rightarrow \underline{y}\textcircled{y}
\end{aligned}
$$

Fig. 1. Input grammar G

with the right hand side of every production. This transformation is repeated until there is no original Nonterminal left. Additionally whenever a Nonterminal N is unfolded the original right hand side is left in the rule with N replaced by a Pseudoterminal of the same name. With the rules given above a production $M \rightarrow NZ_k$ would be transformed into $M \rightarrow \alpha Z_1 Z_k$ and $M \rightarrow \underline{N}Z_k$ with $\underline{N}$

being the newly generated Pseudoterminal. Again this transformation does not change the language of the grammar.
When every original Nonterminal has been unfolded, the unfold transformation

$$
\begin{array}{lll}
Z_3 \rightarrow EZ_{\#} & Y \rightarrow \underline{y}Z_9 & Z_6 \rightarrow \underline{Z_7} \\
Z_{\#} \rightarrow \textcircled{\#} & Z_0 \rightarrow \underline{+}Z_{10} & Z_7 \rightarrow \textcircled{a} \\
E \rightarrow VZ_4 & Z_1 \rightarrow VZ_{11} & Z_8 \rightarrow \textcircled{x} \\
A \rightarrow \underline{(}Z_5 & Z_2 \rightarrow AZ_{12} & Z_9 \rightarrow \textcircled{y} \\
V \rightarrow X & \quad\mid \textcircled{e} & Z_{10} \rightarrow Z_1 \\
\quad\mid Y & Z_4 \rightarrow Z_0 & Z_{11} \rightarrow Z_2 \\
X \rightarrow \underline{x}Z_8 & Z_5 \rightarrow EZ_6 & Z_{12} \rightarrow \textcircled{e}
\end{array}
$$

Fig. 2. Normalized grammar G

is applied repeatedly together with the well known operations for left factorization and the removal of leftrecursion until every production starts with either a (Pseudo)Terminal or Action symbol. From this form the first and only language modification is done: Every rule is cut off after the first occurrence of a Nonterminal, yielding a grammar in which every rule has either the form: $Z_i \rightarrow \underline{t}Z_j$ or $Z_k \rightarrow \textcircled{n}$.
At this point one can easily create a LR parse table from the grammar: Every Nonterminal represents a state and its productions the state transitions. The former yield *shift* transitions and the latter *reduce* operations (the exact operation has been created when the Action symbol was introduced in the first place). The state that is to enter after a *reduce* operation is determined by the productions starting with Pseudoterminals (in those states that are on top of the state stack after the *reduce* was invoked).
Of course one has to populate the Action symbols with their respective lookahead before the parse table can be created but this can be done by examining the rules starting with Pseudoterminals and the rule elements that were cut off before.
Figure 1 shows an example grammar G. The transformation result of the normalization is shown in figure 2. Finally figure 3 shows the parse table as it was derived from the viable prefix grammar.

1.4 Incremental Parsing

In classic applications in compilers or interpreters a parser (called 'batch' parser) is invoked exactly once per input file. The produced AST is then handed over to next stage tools (e.g. semantic analysis) and will not be used again. Anyway it should be clear that in most scenarios the change that was made to an input file between two subsequent parser runs is very small. (Tools like `diff` and `patch` make use of that fact in diverse distributed software development processes.)
So to shorten the time of the parse run it seems natural to re-use the AST from

	$\underline{x}$	$\underline{y}$	$\underline{A}$	$\underline{X}$	$\underline{Y}$	$\underline{E}$	$\underline{(}$	$\underline{)}$	$\underline{+}$	$\underline{\#}$
$Z_{\#}$										(#)
Z_3	Z_8	Z_9		Z_4	Z_4	$Z_{\#}$				
Z_4									Z_{10}	
Z_5	Z_8	Z_9		Z_4	Z_4	Z_6				
Z_6								Z_7		
Z_7								ⓐ		ⓐ
Z_8							ⓧ	ⓧ	ⓧ	ⓧ
Z_9							ⓨ	ⓨ	ⓨ	ⓨ
Z_{10}	Z_8	Z_9		Z_{11}	Z_{11}					
Z_{11}			Z_{12}			Z_5	ⓔ			
Z_{12}								ⓔ		ⓔ

Fig. 3. Parse table

the latest invocation of the parser and only apply the small change to it. This implies that an incremental parser gets two input parameters: The list of input tokens (with the information which tokens represent the change) and an AST that was derived from the same list of input tokens without the change.
For general parsers it needs to be mentioned that the actual advantage of recycling an old AST depends on the grammar and the position of the changed tokens. Theorem 1 gives an example for a situation where incremental parsing yields no advantage at all.

Theorem 1. *The worst-case-complexity of an incremental parser is the same as of a batch parser.*

Proof. *Let G_1 and G_2 be two distinct grammars with $G_1 = (N_1, T, P_1, S_1)$, $G_2 = (N_2, T, P_2, S_2)$ and $N_1 \cap N_2 = \emptyset$. The combined grammar $G_3 = (N_1 \cup N_2, T \cup \{\underline{g_1}, \underline{g_2}\}, P_1 \cup P_2 \cup \{S \to \underline{g_1}S_1, S \to \underline{g_2}S_2\}, S)$ will then accept the language of both grammars (with the Terminal $\underline{g_1}$ or $\underline{g_2}$ as first symbol).*
If an input $\underline{g_1}e, e \in T^*$ with $e \in \mathbb{L}(G_1) \wedge e \in \mathbb{L}(G_2)$ is changed into $\underline{g_2}e$, no element of the abstract syntax tree could be reused since N_1 and N_2 are disjoint, which means the same operations are needed as if a batch parser was invoked. □

Although there is no general advantage for every given grammar over a batch parser it is clear that the incremental parser will perform much better when the change only affects a single node in the syntax tree. We will call a change *constrained* by a Nonterminal if the difference between the old AST and the new one is limited to a node that was introduced during a *reduce* operation for that Nonterminal. It is obvious that such a Nonterminal exists for every change since every grammar as a special start symbol S.
Though this approach obviously can save some parsing time it needs still another motivation: In a classic setup one could easily think of an incremental parser that would its resulting AST in a temporary file so it could be reused. But since reading and decoding the AST would also take $\mathcal{O}(n)$ with n the count of input tokens (the same runtime class a batch parser would typically need) there

would be no point in recycling any data whatsoever. This changes with modern integrated development environments (IDEs).

1.5 Application in the Eclipse IDE

Current IDEs like the Eclipse Java Development Tools (JDT) support the user with mighty tools for refactoring, error detection and automatic code completion. All those operations need the current input text to be parsed into an AST. Since this AST needs to be updated after each change as fast as possible (in fact this process called *reconciling* is invoked 500ms after each change per default) Eclipse delivers the perfect rationale for the implementation of an incremental parser. In addition the IDE will always know the changed region of the text since it has to be displayed in the first place. This allows a slightly faster lexing process (which basically bisects to the changed place, lexes the new tokens and reuses old tokens when possible).
When applied to an Eclipse editor an incremental parser also needs to fulfill two additional requirements: Firstly it should be as less space consuming as possible. Since every open file is represented as an AST in-memory any additionally bytes per node might render the whole IDE useless for real application scenarios. Secondly and less prominent, the parser should literally re-use the old AST from the root. Since Java is an object oriented language the models that are used inside the Eclipse API are often used to store additional informations and states among them (e.g. the expanded state in a tree view). Re-creating similar objects after every change would mean to loose those extra informations and be very irritating to the user.

2 Implementation of an Incremental Parser

A first naive approach on incremental parsing with a shift-reduce parser would be to save the configuration of the parser whenever a token has been shifted and to associate that configuration with the token. In the next run one could easily get the last unchanged token and re-create the configuration in constant time. Although such an algorithm would be relatively easy to implement and work very fast it has two major drawbacks: It does not cover the AST nodes that where built from tokens *behind* the change and, even worse, it wastes an enormous amount of memory. So this kind of algorithm (as it is described in [1] - although optimized there) would clearly fail our above mentioned objective.

2.1 Basic Concept

Although that idea (called ”state matching”) is useless for the application in IDEs its review introduces the concept of splitting the AST into nodes that are before the change and nodes behind it. More formally a node is called ”before” the change if the action that constructed its right-most leaf removed only tokens

from the token stack that precede the changed tokens in the input list and called "behind" when the tokens succeeded the change for its left-most leaf. Those nodes can be reused as seen from theorem 2.

Theorem 2. *Let* $([S_1, \ldots, S_n\$], [w^{-1}v^{-1}\$], [N_1 \ldots N_n\$])$ *be a configuration that PaGe can enter and* $A \Rightarrow^* v$ *a production from the original grammar (and* w *prefixed with the lookahead for this production), then, if there is a state* S_p *with* $GOTO(S_n, \underline{A}) = S_p$*, PaGe will enter the configuration* $([S_1, \ldots, S_n, S_p], [w^{-1}], [N_1, \ldots, N_n, A])$

Proof. From the existence of S_p *follows, that* $S_n \Rightarrow AS_p$ *must be part of the normalized grammar (because otherwise the Pseudoterminal* $\underline{A}$ *would not be in the* $GOTO$ *table for* S_n*).*
Since all following shift/reduce operations and therefore the next states are determined by the lookahead (PaGe is deterministic) and $A \Rightarrow^* v$ *(with* w *being a valid lookahead), it follows that after* $|w|$ *shift operations and a finite amount of reductions (because in any given state there can only be a finite amount of reductions and there are only a finite amount of states on the stack) the parser will finally reduce* v *to* A *and enter the configuration* $([S_1, \ldots, S_n, S_p], [w^{-1}], [N_1, \ldots, N_n, A])$ *after finding* $S_p = GOTO(S_n, \underline{A})$ □

Obviously a third group of nodes (including the root node) exists, in which every token "overlaps" the change and thus has to be parsed for changes. Following this observation [6] presents another approach with no additional space requirements. The algorithm described here is similar to this optimal solution but makes use of PaGe's special features and has a much simpler design since it only processes one change at a time (which is caused by the eclipse environment which will always report changes one after the other).
The splitting of the AST naturally implies a tree-phase algorithm: Firstly recreate the state just before the changed tokens would be shifted by descending the "before" part of the AST, after setting the parser into its initial configuration. Since every node A covers a word v that has not been changed there must be a production $A \Rightarrow^* v$ in the grammar and theorem 2 applies (the validity of the configurations follows directly from the theorem and the fact that we start with the initial configuration that is always valid). So the parser is effectively short-circuited instead of going through all configurations. Secondly to handle the changed part the algorithm only invokes the parser until the constraining node has been restored and finally the algorithm reuses the behind part by ascending the AST from the input tokens.
The only problem that remains unsolved with this algorithm is the calculation of the constraining node. Fortunately this is not necessary when the second and third phase are merged together: The third phase ascension is always applied if possible (that is: the next input tokens are unchanged and their parent AST nodes match a Pseudoterminal that occurs in the *GOTO* set of the current state). So every time a normal parse step was applied successfully and the next Terminal would be shifted, this Terminals ancestor nodes (it has none if it is

affected by the change) are checked for direct reduction and the highest one that completely lies before the change is reused in a short circuit reduction similar to the one in phase one. Afterwards a batch parser step is invoked and so on.
This eliminates the need for an explicit calculation of the constraining node and also allows the recycling of nodes that had been created from a different parser configuration or are children of the constraining node - that way the incremental parser is reuses even more nodes than any state matching parser.

2.2 Corner Cases

Although the above algorithm is correct (it returns the same AST that the batch parser would return), it needs to be refined to correctly handle some corner cases which PaGe can handle in batch mode. The most obvious is the lookahead problem: Since PaGe is a *LR(n)* parser generator, the restriction in theorem 2 to a valid lookahead v can be very important. Since the current PaGe implementation generates lookahead only where needed, one could determine the used lookahead in every step of the incremental parser and include this into the validity check for the re-usage of a node. This method was not yet implemented due to the consideration of the expected additional costs. Instead the change will be expanded by the maximum length of lookahead used. With this simple step it is made sure that no token has changed that was used as a lookahead.
The next corner case that had to be handled was the existence of epsilon productions. Though PaGe's meta grammar does not allow direct epsilon productions, they are still possible by using the optional construct for the right hand side of a production (e.g $A \rightarrow [B]$). This forces the incremental parser to determine if an empty AST node E, that effectively has no Terminals as leafs, is affected by a given change (Just stopping the process as if the change was reached would also work but be very ineffective). Since the incremental parser also can not recreate the configuration that the batch parser had E was pushed the only possible solution is to simulate the batch parser by shifting tokens until a state S_f is reached with $\underline{E} \in GOTO(S_f)$. This is not needed in the second phase since the batch parser will recreate any empty nodes that are not directly reused.
Finally, the error handling mechanisms pf PaGe had to be handled. In the current implementation, PaGe will always skip erroneous tokens and not try to add any. (This is due to the fact that most languages are defined by lists of elements and the parser can easily ignore one of those elements while the AST structure remains correct.) This process is called *recovering.*
Since the incremental parser will get erroneous input quite often, it needs to handle it as well as the batch parser - only faster. To simplify this the first handling is left to the lexer: If there are old syntax errors far (that is: more than the used lookahead) tokens away from the change, the error tokens are hidden from the input. Otherwise the error and the change are simply merged. If the batch parser encounters an error in the second phase the error handling is left to it, since after a successful recovery the incremental parser can simply continue. Obviously there is some room for improvement here: The incremental parser could

check the AST for some sense of what element actually constrains the syntax error, but this remains object to further development.

```
private final void incrementalPhase1(Absy node, Range change) {
/*
 * before the change spot we can simply short circuit the shift/reduce
 * cycle
 */
  if (change.isBehind(node.getRange())) {
    reduce(node);
    skipInput(node);
  } else {
    // do they overlap?
    if (!node.getRange().isBehind(change)) {
      for (Absy child : node.getChildren()) {
        if (child.getRange().getStartToken() >= change.getStartToken())
          break;
        if (child.getRange().getStartToken() >= 0) {
          shiftIntermediateTerminals(child);
          incrementalPhase1(child, change);

          if (child.getRange().getEndToken() >= change.getStartToken())
            break;
        } else {
          /* handle empty productions */
          if (change.isBehind(parser.getInputStack().peek().getRange()))
            handleEpsilonAbsy(change, child);
        }}}}}
```

Fig. 4. Incremental Phase 1

2.3 Implementation and Tests

The incremental parser was implemented as an additional module that can be used with PaGe to not interfere with the current development (Especially to not introduce any new bugs).
Figure 2.2 shows the implementation of the first phase: Nodes that are before the change are reduced, others are descended and their children are handled, if they do are not completely behind the change.
The second phase implementation in figure 2.3 shows how the ascension is handled: After all possible actions are applied (to make sure the next token is really about to be shifted), the highest parent node that can be reused is put on the token stack effectively short circuiting a reduction. For the testing process using JUnit was an obvious choice. The test setup works on the library files from MOSILAB [4], an implementation of the Modelica language. For every source

```
boolean incrementalPhase2(Token token, Range change) throws ParserException {
  applyAllActions();

  Absy highest = getHighestValidParent(parser.getStateStack().peek(),
      token, change);
  if (highest != null) {
    // clean up any already shifted tokens
    for (int i = highest.getRange().getStartToken(); i < token
        .getRange().getStartToken(); i++) {
      parser.getTokenStack().pop();
      parser.getStateStack().pop();
    }
    reduce(highest);
    // remove the input tokens, we don't need anymore
    skipInput(highest);
    return true;
  }
  return false;
}
```

Fig. 5. Incremental Phase 2

file a Test case is created that runs the incremental parser after marking some terminals as changed and compares the output AST with that from the batch parser. For every iteration of that process the change is moved roughly 1% ahead on the input. Although the coverage may be bad since no real change is handled, this method has the benefit of allowing a good benchmark of the incremental parsing for constant changes.

2.4 Runtime and Space Complexity

As shown in Theorem 1 there can be no general advantage for every given grammar (or every given AST). So for any given change the constraining node C the runtime will not go below $\mathcal{O}(l_C)$ with l_C denoting the amount of input tokens covered by C.
Since in the first phase the tree is descended, this phase finishes in $\mathcal{O}(log(n))$ if we assume the AST being balanced with n nodes. Additionally after C has been reconstructed all of its right siblings (and its parents right siblings and so on) can be reused. Since the algorithm always searches for the *highest* reusable parent node in the second phase, the search for the highest valid parent will return nodes closer to the AST root at least all $\mathcal{O}(c_{max})$ steps with c_{max} being the maximum of children a node can have in the AST. Because the maximum distance a node can have in the AST is $\mathcal{O}(log(n)$, this leads to a runtime of $\mathcal{O}(c_{max}log(n)^2)$ for the second phase. Note, that c_{max} is not really a constant due to the existence of lists of AST nodes. Yet this makes the tree more balanced. But for any practical applications (since the parser shall work in IDEs for humans) one can assume

the length of those lists as being limited, so the runtime of $\mathcal{O}(log(n)^2 + l)$ with l being very small for a real programming language is archived.
The algorithm was designed with the target of zero additional space cost in mind. That is archived with one minor exception: In the AST that is produced by PaGe every node has a direct parent reference. Without this reference the AST could still be used, but in the second phase to determine the path from the root node to the next unchanged token additional runtime cost of $\mathcal{O}(log(n)$ would be required, not changing the complexity class but effectively halving the incremental parsers performance in that phase.

3 Optimizations

Although the runtime complexity is near to optimal (see [6]) and only non optimal because of some properties of PaGe or the input grammar, there is still room for some optimizations:

3.1 Sibling Ascension

After the constraining node C has been parsed, the complete right hand side of the AST can be reused by the incremental parser. This means that instead of walking up the tree from the bottom, one could reuse the right hand side siblings of C directly. Then the sibling of Cs parent and so on.
Since C is not calculated explicitly, we implemented this again implicitly by checking if the siblings of the last reused node in phase two can be reused too. Again this has the advantage that also children of C can be reused.
Obviously this speeds up the AST ascension in the second phase. Instead of ascending one level in $\mathcal{O}(c_{max}log(n))$ with this optimization it is done in $\mathcal{O}(c_{max})$. Therefore the overall runtime is $\mathcal{O}(l_C + log(n))$ with Sibling Ascension enabled. Although this seems to be a big performance gain, benchmarks (table 6) did not support that assumption. In fact tested grammars showed a runtime of $\mathcal{O}(l_C + log(n))$ in the above mentioned setup.

3.2 Lists

Another difficulty are the Lists that are used by PaGe: While grammar productions like $A \rightarrow B^*$ are transformed into recursive productions internally the created actions form special tokens that expose a list interface instead of the degenerated trees one would expect. Although the parser works with those lists (they can be reconstructed in the second phase or completely reused in the first phase), there is still some room for optimizations: The deconstruction of a list could work by bisecting it and since it is clear that a list *must* be created in the incremental run, if the first child covers more tokens than the maximum lookahead one could try to split the list in a prefix and suffix part to speed up the recreation. This is not yet implemented since that change might demand changes of the batch parser which was not allowed by the current development

concept.

3.3 Object Reusage

While the algorithm meets all requirements in runtime and space that were mentioned above, its behaviour is suboptimal in the aspect of object reusage: Though all nodes that have no changes are recycled (and some more that changed their place inside the AST), the path from the AST root to the constraining node C is completely recreated even if it is not altered structurally.
Again one does not know this path because of the implicit calculation of C, but it can only consist of nodes that were visited but not reused during the first phase. Since every Action that the parser applies to the stack "knows" how much Tokens it will remove from the Token Stack, it can easily check if the leftmost of these tokens already has a parent and if this parent matches the Class that this action would instantiate. If this is the case, instead of creating a new AST node, that parent can be reused and populated with the new set of child nodes. Since the parser creates the AST bottom up, that way every node that has children in the "before" part of the AST will be reused, if possible.

Tokens	Batch (ms)	Incremental (ms)	Optimized (ms)
4906	74.3	1.9	5.5
5181	88.3	3.3	3.3
7423	105.9	4.5	3.6
8871	141.5	5.7	11.2
10046	131.3	7.4	7.6
10088	148.8	4.9	6.6
11212	181.5	5.0	5.2
14385	178.1	4.1	4.3
15720	188.8	4.4	5.6
17208	236.8	6.7	4.7
19568	227.9	5.8	4.8
20295	263.8	3.5	3.6
21861	259.8	5.5	7.1

Fig. 6. Benchmarking results

3.4 Benchmarks

As mentioned above we tested the incremental parser by marking single input tokens as changed (this can be easily done by deleting their parent reference). Therefore the effect of l_C is minimized, making this benchmark somewhat synthetic (one could consider this a realistic scenario for refactoring operations

where e.g. identifiers are renamed).
The table in figure 6 shows the worst case runtime of the incremental parser and the batch parser against the number of tokens in the file.
To minimize the influence of dynamic optimizations inside the JVM every benchmark was run ten times.
Surprisingly the optimization did not yield a significant advantage but some major performance drawbacks. Obviously the overhead of the implementation surpasses the advantage of ascending the tree in an optimized order. (Also it should be clear that the pure tree ascension is extremely fast compared with the parsing operations and reusable checks).

4 Example

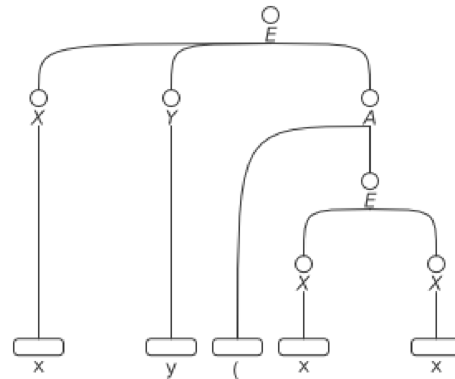


Fig. 7. Example AST for G

As an example for the incremental parser consider the AST for G shown in figure 7. This tree was constructed from the input `x + y (x + x)`. If this input is changed to `x + x (x + x)`, the parser will enter the following configuration after the first phase:

$$([Z_3, Z_4, \$], [\underline{)}, \underline{x}, \underline{+}, \underline{x}, \underline{(}, \underline{+}, \underline{x}\$], [X\$])$$

The parser will then parse the $\underline{+}$ and $\underline{y}$ Terminals, entering:

$$([Z_3, Z_4, Z_{10}, Z_{11}\$], [\underline{)}, \underline{x}, \underline{+}, \underline{x}, \underline{(}\$], [X, \underline{+}, X\$])$$

The next Terminal, $\underline{(}$ is the leftmost Terminal of the A node which can then directly be recycled:

$$([Z_3, Z_4, Z_{10}, Z_{11}, Z_{12}\$], [\$], [X, \underline{+}, X, A\$])$$

And after a final reduction reaching the final state $Z_{\#}$:

$$([Z_3, Z_{\#}\$], [\$], [E\$])$$

In this example the only newly created node would be the new X node, since E could be recycled by the above mentioned object reusage mechanism.

5 Conclusion

The proposed algorithm makes it possible to reconcile models from source code in real time for arbitrary LR grammars. This allows IDE developers to create sophisticated tools without having to struggle with complex tree algorithms.
The benchmarks taken from a real programming language source code reveal that optimizations on the algorithm do not make a notable difference in the time a reconciling operation takes. Therefore the tradeoff of additional complexity and possible sources of errors must be considered when using the parser for a given language.
This algorithm will be implemented for a IDE supporting the MOSILAB language and revised based on the experiences from this application. Afterwards the algorithm may be merged into PaGe.

References

1. Agrawal, R., Detro, K.D.: An efficient incremental lr parser for grammars with epsilon productions. Acta Inf. 19(4), 369–376 (1983)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
3. Kunert, A.: Lr(k)-analyse für pragmatiker (2008), http://www.informatik.hu-berlin.de/ kunert/papers/lr-analyse/, (de)
4. Nytsch-Geusen, C., Ernst, T., Nordwig, A., et al.: Mosilab: Development of a modelica based generic simulation tool supporting model structural dynamics. In: Schmitz, G. (ed.) Proceedings of the 4th International Modelica Conference, Hamburg, March 7-8, 2005. pp. 527–535. TU Hamburg-Harburg (2005)
5. P.Pepper, M.: Compilergenerator für opal-2 (2008), (unpublished)
6. Wagner, T.A., Graham, S.L.: Efficient and flexible incremental parsing. ACM Transactions on Programming Languages and Systems 20 (1996)

Generierung von Hyperkantenersetzungsgrammatiken zur Heapabstraktion

Christina Jansen[1] und Jonathan Heinen[2]

Lehrstuhl für Informatik 2, RWTH Aachen

Zusammenfassung. Bei der Verifizierung von heapbasierten, dynamischen Datenstrukturen stellen unendliche Zustandsräume ein Problem dar. Die potentiell unendliche Menge von Heapzuständen läßt sich durch Abstraktion endlich repräsentieren und mithilfe dieser Repräsentation durch Techniken wie Model Checking verifizieren. Grundlage dieser Abstraktion sind Hyperkantenersetzungsgrammatiken, wobei die Form einer geeigneten Grammatik stark von der zugrundeliegenden Datenstruktur und der Anwendung abhängt. Erfüllt sie dabei alle notwendigen Eigenschaften bezeichnet man sie als Heapabstraktionsgrammatik. Wir möchten vorstellen, wie die Konstruktion einer solchen Heapabstraktionsgrammatik abläuft und wie das Lernen von Produktionsregeln während der Programmausführung aussehen kann.

1 Einleitung

Heapbasierte Datenstrukturen spielen in den heutigen Programmiersprachen eine wichtige Rolle. Dadurch, dass zur Laufzeit Objekte erstellt oder entfernt werden können, und damit die Menge der Heapzustände im Allgemeinen unendlich und zur Kompilierzeit unabsehbar ist, stellen sie für die meistgenutzten Verifikationstechniken ein Problem dar. Man ist daher bemüht potentiell unendliche Strukturen durch endliche Abstraktionen zu repräsentieren ohne relevante Informationen zu verlieren.

Eine Möglichkeit zur Abstraktion von Heapzuständen bietet das Konzept der Hyperkantenersetzungsgrammatiken (HRGs) [1]. Dabei wird ein Heapzustand durch einen Hypergraphen repräsentiert, die Produktionsregeln der Grammatik werden zum Abstrahieren und Konkretisieren der Hypergraphen angewandt. Die grundlegende Idee besteht darin, die Rückwärtsanwendung von Produktionsregeln zuzulassen, diesen Vorgang bezeichnen wir als Abstraktion. Falls notwendig können abstrahierte Teilgraphen durch normale Regelanwendung wieder konkretisiert werden. Dies hat zur Folge, dass keine Semantik für die abstrakten Teile der Graphen gebraucht wird, da Operationen nur auf konkreten Teilgraphen ausgeführt werden müssen. [2]

Zur korrekten Abstraktion von Heapzuständen muss eine HRG einige notwendige Eigenschaften erfüllen, damit z.B. gewährleistet wird, dass keine relevanten

Informationen verloren gehen etc. Die rückwärtige Nutzung von Produktionsregeln führt zusätzlich zu weiteren Anforderungen an die Grammatik, damit Dinge wie Terminierung oder die Existenz entsprechender Konkretisierungsregeln garantiert werden können. Für die Abstraktion geeignete HRGs nennen wir Heapabstraktionsgrammatiken. Die Konstruktion einer solchen zu einer gegebenen Grammatik läßt sich automatisch durchführen, die Idee dabei ist es, die HRG in eine spezielle Normalform zu bringen aus der sich die gewünschten Eigenschaften durch einfache Schritte ableiten lassen.

Eine Frage bei diesem Ansatz der Heapabstraktion ist weiterhin die Wahl einer geeigneten Eingabegrammatik. Denn es existiert keine universelle Heapabstraktionsgrammatik, vielmehr hängt diese stark von dem betrachteten Problem ab und muss individuell für dieses erstellt werden. Man hat demnach verschiedene Möglichkeiten eine HRG zu bestimmen: manuell durch genaue Betrachtung der Datenstruktur und der zu verifizierenden Anwendung, Inferenz einer Grammatik durch Eingabe einer Beispielmenge von Graphen und Lernen von Grammatikregeln während der Programmausführung.

Sowohl die Konstruktion einer Heapabstraktionsgrammatik anhand einer speziellen Normalform als auch den Ansatz des Lernens einer HRG durch Kombination der drei oben genannten Möglichkeiten, möchten wir hier vorstellen.

2 Hypergraphen & HRGs

Bevor wir zu Themen wie Heapabstraktion und dem Lernen von HRGs kommen, betrachten wir vorerst die Grundlagen von Hypergraphen und der Kantenersetzung.

2.1 Hypergraphen

Hypergraphen stellen eine Generalisierung der weit verbreiteten, "normalen" Graphen dar. Formal lassen sie sich durch ein 5-Tupel (V, E, lab, att, ext) beschreiben, wobei sie wie gewohnt Knoten V und Kanten E besitzen. Letzteren wird durch die Funktion $lab : E \to \Sigma$ eine Bezeichnung aus dem Alphabet Σ zugewiesen. Kanten in Hypergraphen dürfen eine beliebige Anzahl von anliegenden Knoten besitzen, die Attraktor-Funktion $att : E \to V^*$ ordnet dabei jeder Kante die zugehörigen Knoten zu. Zuletzt dürfen Knoten als extern gekennzeichnet werden, sie werden durch die ext-Menge beschrieben und ihre Anzahl bestimmt den Rang des Hypergraphen.

Eine graphische Darstellung eines Hypergraphen findet sich in Abb. 1. Er besitzt drei Knoten, wobei einer als extern markiert ist. Außerdem enthält er zwei Hyperkanten, eine Terminalkante (hier als übliche gerichtete Kante dargestellt) mit der Beschriftung n und eine Nichtterminalkante, die mit X bezeichnet ist.

2.2 HRGs

Eine Hyperkantenersetzungsgrammatik (engl. hyperedge replacement grammar) ermöglicht das Ableiten von Terminalgraphen durch gezielte Ersetzung von Hy-

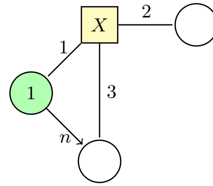


Abb. 1. Beispiel-Hypergraph

perkanten. Sie wird analog zu String-Grammatiken über ein 4-Tupel $G = (N, T, P, S)$ formalisiert, dessen Nichtterminalsymbole N und Terminalsymbole T das Alphabet $\Sigma = N \cup T$ bilden mit $T \cap N = \emptyset$. Die Produktionsregeln $(X, H) \in P$ sind auf der linken Produktionsregelseite durch Nichtterminale $X \in N$ beschriftet und enthalten einen Hypergraphen H auf der rechten Regelseite. Sie werden angewandt indem eine mit X beschriftete Hyperkante durch H ersetzt wird. Dabei werden die externen Knoten von H auf die an der Hyperkante anliegenden Knoten abgebildet. Einen solchen Ableitungsschritt nennen wir Kantenersetzung.

Abbildung 2 zeigt eine HRG für einfach verkettete Listen, denn ihre Sprache – die Menge der ableitbaren Terminalgraphen – enthält alle Listen, in denen jedes Objekt einen Zeiger auf seinen Nachfolger besitzt. Eine beispielhafte Ableitungsfolge, die eine solche Liste zum Ergebnis hat, ist in Abb. 3 zu finden.

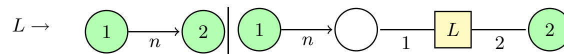


Abb. 2. HRG: einfach verkettete Liste

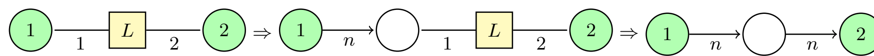


Abb. 3. Kantenersetzung: einfach verkettete Liste

3 Heapabstraktionsgrammatiken

Wie bereits erwähnt lassen sich HRGs als Abstraktionsmechanismus für eine Menge von Heapzuständen nutzen. Betrachten wir genauer wie Heapzustände als Graph dargestellt und durch die Produktionsregeln der HRG abstrahiert werden.

3.1 Heaprepräsentation

Wir repräsentieren Heapzustände durch Hypergraphen, indem wir Variablen und Zeiger durch entsprechende Kanten darstellen. Ein solcher Hypergraph wird dann Heapkonfiguration genannt. In ihm stellen Terminalkanten vom Rang 2 Zeiger dar und Terminalkanten vom Rang 1, die mit Variablen beschriftet sind, entsprechen den Variablen im Heap. Eine Heapkonfiguration mit einer Variablen x und Selektoren n ist in Abb. 4 dargestellt. Programmanweisungen, die den Heap manipulieren, werden analog im Hypergraphen durchgeführt. Eine Anweisung zur Löschung eines Objektes ist dabei nicht vorgesehen, dies geschieht stattdessen durch eine Zuweisung von nil. Abbildung 5 zeigt die Umsetzung der wichtigsten Programmanweisungen im Hypergraphen.

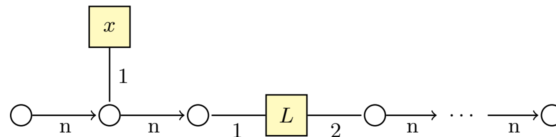


Abb. 4. Heapkonfiguration

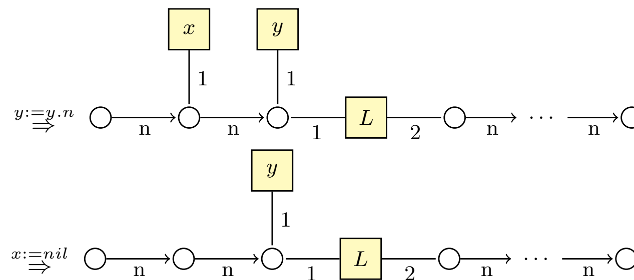


Abb. 5. Programmanweisungen

3.2 Heapabstraktion

Als weiterer Schritt soll nun eine Menge von Heapzuständen durch eine Heapkonfiguration dargestellt werden. In Abbildung 4 haben wir schon eine Heapkonfiguration, in der bereits Teile eines Heaps durch die Hyperkante mit dem Bezeichner L abstrahiert wurden, gesehen. Nichtterminalkanten repräsentieren immer einen abstrakten Teilgraphen und damit Teile eines Heaps. Ein Abstraktionsschritt entspricht dabei einer rückwärtigen Produktionsregelanwendung auf

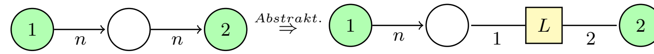


Abb. 6. Heapabstraktion

einer Heapkonfiguration. Legen wir die HRG aus Abb. 2 zugrunde, ist ein Abstraktionsschritt in Abb. 6 zu sehen.

Wie bereits erwähnt führt die Zulässigkeit der Rückwärtsanwendung von Produktionsregeln zu einigen problematischen Ableitungen, die allerdings durch zusätzliche Anforderungen an die HRG vermieden werden können. Eine zulässige Heapabstraktionsgrammatik muss die folgenden Eigenschaften besitzen:

- Produktivität
- Wachstum
- Variablenfreiheit
- Typisierung
- Lokale Apex-Eigenschaft

Produktivität wird gefordert, damit keine Hypergraphen abstrahiert werden können, die später nicht mehr zu Terminalgraphen konkretisiert werden können. Wachstum der Produktionsregeln wird benötigt um die Terminierung von Abstraktionsschritten gewährleisten zu können. Die Variablenfreiheit stellt sicher, dass keine wichtigen Informationen über Programmvariablen bei der Abstraktion verloren gehen, denn sie verbietet das Vorkommen von Variablenbezeichnern in rechten Produktionsregelseiten. Die Typisierung der Grammatik stellt sicher, dass aus mehrfachen Ableitungsschritten keine unzulässigen Hypergraphen entstehen. Sie fordert, dass jedes Nichtterminal einen festen Rang und feste ausgehende Terminalkanten an den externen Knoten besitzt. Die lokale Apex-Eigenschaft gewährleistet, dass immer, wenn ein Konkretisierungsschritt nötig ist, eine geeignete Produktionsregel existiert, die innerhalb eines Ableitungsschrittes an der gewünschten Stelle einen konkreten Teilgraphen generiert. Die Problematik, sollte dies nicht der Fall sein, läßt sich an einem kurzen Beispiel gut veranschaulichen.

Beispiel 1. Betrachten wir wiederum die in Abb. 7 diesmal leicht modifizierte HRG für einfach verkettete Listen sowie die Heapkonfigurationen aus Abb. 5. Soll nun an dem Knoten, an dem die Variablenkante y anliegt, konkretisiert werden – weil z.B. $z := y.n$ ausgeführt werden soll – ist die einzige am betreffenden Knoten konkretisierende Produktionsregel die Terminalregel der Grammatik. Wird diese jedoch angewandt, lassen sich mit der resultierenden Heapkonfiguration keine Listen beliebiger Länge mehr ableiten. Es muss daher durch eine Eigenschaft sichergestellt werden, dass immer eine geeignete Produktionsregel zur Konkretisierung vorliegt.

Von diesen fünf Eigenschaften lassen sich Produktivität, Variablenfreiheit und Typisierung relativ leicht sicherstellen. Die verbliebenen zwei – Wachstum

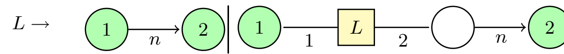


Abb. 7. Heapabstraktion

und lokale Apex-Eigenschaft – erfordern eine komplexe Konstruktion um eine äquivalente Grammatik mit diesen Eigenschaften zu erhalten. Die Idee dieser Konstruktion beruht darauf, die Eingabegrammatik in eine Normalform zu bringen, anhand derer es dann mit einfacher Kantenersetzung möglich ist eine Heapabstraktionsgrammatik abzuleiten. Einen Ansatz zur Konstruktion von äquivalenten HRGs mit Apex-Eigenschaft wird in [3] vorgestellt. Dieser fordert einerseits eine zu starke Eigenschaft, was zu unnötig grossen Regeln führt, da wir lediglich lokale Apex-Eigenschaft benötigen. Andererseits läßt er sich nur für Grammatiken anwenden, deren Sprachen beschränkt sind. Da aber Datenstrukturen wie z.B. gewurzelte Bäume oder Listen mit Zeigern auf den Wurzelknoten darstellbar sein sollen, ist diese Konstruktion für unsere Zwecke nicht ausreichend. Die Idee läßt sich allerdings auch auf für die von uns benötigte lokale Apex-Eigenschaft mit einigen Anpassungen übertragen. Sie stellt weiterhin implizit auch Wachstum der Grammatik sicher. Auch für unseren angepassten Ansatz läßt sich die Gleichheit der Sprachen von Eingabe- und resultierender Grammatik zeigen.

4 Lernen von HRGs

Wie wir bereits gesehen haben, lassen sich HRGs zur Verifizierung von dynamischen, heapbasierten Datenstrukturen einsetzen. Weiterhin ist es auch in vielen Fällen möglich zur Eingabegrammatik eine äquivalente, zulässige Heapabstraktionsgrammatik zu konstruieren. Eine Herausforderung besteht vor jedem Verifikationsprozess noch darin, eine entsprechende Eingabegrammatik zu bestimmen.

Ein Ansatz zur Inferenz einer HRG aus einer Beispielmenge von Graphen ist in [4] zu finden. Er setzt sich zusammen aus vier Basis-Funktionen – INIT, RENAME, DECOMPOSE und REDUCE –, die in beliebiger Reihenfolge und beliebig oft aufgerufen werden, um Produktionsregeln abzuleiten. Man kann zeigen, dass die so entstehenden HRGs kontextfrei sind und alle Graphen aus der Beispielmenge erkennen und damit einer Überapproximation entsprechen. Der Algorithmus ist allerdings hochgradig nichtdeterministisch und damit für unsere Anwendung nicht direkt nutzbar.

Um eine deterministische Variante zu erhalten, haben wir einige Heuristiken eingeführt, wie z.B. das Zusammenfassen von gleichartigen Nichtterminal-Bezeichnern oder die Berechnung des minimalen Schnitts als Bestimmungshilfe für die Produktionsregeln. Weiterhin erlauben wir das Lernen der HRG während der Ausführung. Sobald ein Heapzustand auftritt, der nicht vollständig abstrahiert werden kann, wird er zur Beispielmenge hinzugefügt und damit Produktionsregeln erlernt, die seine Abstraktion ermöglichen. Die Zulässigkeit der HRG wird dabei durch die im vorangegangen Abschnitt vorgestellte Konstruktion her-

gestellt. Ein solcher Ansatz bietet den Vorteil, dass zu Beginn keine Beispielmenge von Graphen als Eingabe geliefert werden muss, sondern diese während der Programmausführung automatisch entsteht. Dies führt dazu, dass alle auftretenden Heapzustände vollständig abstrahiert werden können, denn entsprechende Regeln werden bei Bedarf erlernt. Wir erlauben weiterhin die Eingabe einer HRG zu Beginn, diese ist jedoch nicht mehr zwingend notwendig.

Testläufe, die unsere Version vom Inferenzalgorithmus aus [4] benutzten, erzielten für viele gängige Datenstrukturen wie einfach und doppelt verkettete Listen, binäre Bäume etc. als Resultat eine HRG, die der manuell erstellten Grammatik sehr ähnelte oder sogar vollständig gleich war.

Literatur

1. Annegret Habel: Hyperedge Replacement: Grammars and Languages. Springer-Verlag New York, Inc., 1992
2. Stefan Rieger und Thomas Noll: Abstracting Complex Data Structures by Hyperedge Replacement. Springer-Verlag, 2008
3. Joost Engelfriet, Linda Heyker und George Leih: Context-Free Graph Languages of Bounded Degree are Generated by Apex Graph Grammars. Acta Inf., Vol. 31, 1994
4. Eric Jeltsch and Hans-Jörg Kreowski: Grammatical Inference Based on Hyperedge Replacement. Graph-Grammars and Their Application to Computer Science, 461-474, 1990

Combining Automated Reasoning and Algebraic Methods in Theorema

Tudor Jebelean
RISC, Johannes Kepler Universität Linz

We present some applications of the *Theorema* system to the generation of invariants for imperative loops and to automated proving in elementary analysis, which are based on the interaction of logic techniques with methods from computer algebra and from algebraic combinatorics.

The *Theorema* project (`www.theorema.org`), provides an uniform logic frame for the exploration of mathematical theories – [1], based on automatic reasoning.

The use of combinatorial and algebraic methods in conjunction with automated reasoning leads to powerful analysis tools, because they allow the automatic generation of inductive assertions for programs [4] – joint work with Laura Kovacs. The method generates all the invariants which can be represented as polynomial equations (in fact, a basis for the ideal generated by the corresponding polynomials) in two stages: first the recursive equations corresponding to the evolution of loop variables are transformed into closed formulae (depending on the loop counter) using combinatorial techniques; second these closed forms are used in successive applications of the Buchberger algorithm in order to find out the invariant ideal.

We also show how to significantly enhance the power of automatic provers [5,3] – joint work with Bruno Buchberger and Robert Vajda – in particular for reasoning in numeric domains (reals, integers) by using the CAD method (Cylindrical Algebraic Decomposition) in order to generate natural proofs in elementary analysis (the so called epsilon–delta proofs). Namely, by applying the S-Decomposition [2] logical technique we decompose the original proof problem into several numerical conjectures which involve existential quantifiers, whose witnesses are then found by CAD. This combination of techniques builds a prover with the distinctive feature that it does not need all the axioms of the underlying domain (e.g. the reals), but it automatically finds the appropriate lemmata which are necessary for completing the proof.

References

[1] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 2005.

[2] T. Jebelean. Natural Proofs in Elementary Analysis by S-Decomposition. RISC Report 01-33, November 2001.

[3] T. Jebelean. Using Computer Algebra for Automated Reasoning in the Theorema System, 2005. Invited talk at Seventh Asian Symposium on Computer Mathematics (ASCM 2005).

[4] L. Kovacs and T. Jebelean. Finding Polynomial Invariants for Imperative Loops in the Theorema System. In S. Autexier and H. Mantel, editors, *Proceedings of Verify'06 Workshop, IJCAR'06, The 2006 Federated Logic Conference*, pages 52–67, 2006.

[5] R. Vajda, T. Jebelean, and B. Buchberger. Combining Logical and Algebraic Techniques for Natural Style Proving in Elementary Analysis. Mathematics and Computers in Simulation, 79 (8), pp: 2310–2316, Elsevier, 2009.

Automatic Calculation of Coverage Profiles for Coverage-based Testing ⋆

Raimund Kirner[1] and Walter Haas[1]

Vienna University of Technology, Institute of Computer Engineering,
Vienna, Austria, raimund@vmars.tuwien.ac.at

Abstract. Code-coverage-based testing is a widely-used testing strategy with the aim of providing a meaningful decision criterion for the adequacy of a test suite. Code-coverage-based testing is also used for the development of safety-critical applications, as the modified condition/decision coverage (MCDC) is proposed by the DO178b document.
One critical issue of code-coverage testing is that they are typically applied to source code while the generated machine code may result in a different code structure due to code optimizations performed by an compiler. In this work we describe the automatic calculation of coverage profiles describing which structural code-coverage criteria are preserved by which code optimization. These coverage profiles allow to easily extend compilers with the feature of preserving any given code-coverage criteria by enabling only those code optimizations that preserve it.

1 Introduction

Testing is an established and accepted technique to increase the confidence in the correctness of a computer system. In contrast to formal verification, testing is not aimed to cover the full behavior of the system. But in contrast to formal verification, testing has the strong advantage that it operates on the real operation, including all low-level system details and physical behavior. Formal verification on the other side allways resides at a certain abstraction level, allowing the full behavioral coverage at this abstraction level. Thus, testing and formal verification are complementary approaches, both are necessary for the development of safety-critical systems.

Within this paper we focus on the testing part of verification, addressing the challenges towards portable test-data generation. Derivation or generation of test data is preferably done at the same level where the program is developed, typically a high-level programming language or any modeling environment with automatic code generation. First, this is the preferred way to do if the test data are written manually. Second, this is also beneficial for automatic test-data calculation, as it allows to reduce complexity by taking benefit of the abstract program representation. Thirst, this is preferred for portability issues, like cross-platform testing.

⋆ The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Sustaining Entire Code-Coverage on Code Optimization" (SECCO) under contract P20944-N13.

We assume that test-data generation is guided by structural code coverage, for example, *statement coverage*, *condition coverage*, or *decision coverage*. Structural code-coverage on its own is not a very robust coverage metric for software testing, but it is a useful complementary metric that indicates program locations of weak coverage by test data.

Using source-code based derivation of test data, it is the challenge to ensure that the test data fulfill an analogous structural code-coverage metric at the machine-code level as they achieve at source-code level. We call it analogous code-coverage metric, because several structural code-coverage metrics make no sense at machine-code level, because, for example, the grouping of several conditions to a decision is a source-level concept that is not available at machine-code level. If a compiler performs complex code optimizations that, for example, introduce new paths or change the reachability of some statements [1], this may disrupt the structural code coverage achieved at the original program.

We propose an approach toward the preservation of structural code coverage when transforming the program [2, 3]. For this we use a so-called *coverage profiles*, i.e., a pre-calculated table that specifies for each structural code-coverage metric which code transformations of the compiler guarantee to preserve it. Such a coverage profile can be easily integrated into a compiler such that only those code transformations are enabled that preserve the chosen structural code coverage. The conceptual integration of coverage profiles into a compiler is shown in Figure 1. In this paper we focus on the abstract specification of code transformations and on the calculation of the coverage profiles.

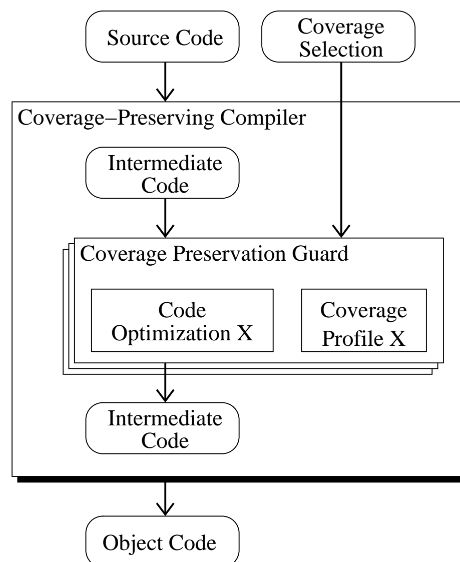


Fig. 1. Application of a Coverage Profile

Besides the functional software testing, the preservation of structural code coverage is also of high interest for hybrid timing analysis, i.e., an approach to determine the tim-

ing behavior of a program based on the combination of execution-time measurements and program analysis [4, 5].

2 Structural Code Coverage for Software Testing

Structural code-coverage criteria are testing metrics to quantify the control-flow coverage of the program for a given set of test data. In this section we describe a few exemplary structural code-coverage metrics to show the calculation of compilation profiles. Formal definitions of some additional structural code-coverage metrics can be found in [3, 6].

2.1 Basic Definitions

In the following we give a list of basic definitions that are used to formally describe properties of structural code coverage and conditions for preserving structural code coverage:

Program P denotes the program before (P_1) and after (P_2) the transformations for which we want to preserve structural code coverage.

Control-flow graph (CFG) is used to model the control flow of a program [7]. A CFG $G = \langle N, E, s, t\rangle$ consists of a set of nodes N representing *basic blocks* (see below), a set of edges $E : N \times N$ representing the control flow (also called control-flow edges), a unique entry node s, and a unique end node t.

Basic block of a program P is a code sequence of maximal length with a single entry point at the beginning and with the only allowed occurrence of a control-flow statement at its end. We denote the set of basic blocks in a program P_i as $\mathrm{B}(P_i)$.

Decision is a Boolean expression composed of *conditions* that are combined by Boolean operators. If a *condition* occurs more than once in the *decision*, each occurrence is a distinct condition [8]. However, the *input* of a decision is the set of its conditions without duplicates. A decision is composed of one or more basic blocks. We denote the set of decisions of a program P_i as $\mathrm{D}(P_i)$.

There are programming languages, where decisions are hidden by an implicit control flow. For example, in ISO *C* due to the short-circuit evaluation the following statement `a = (b && c);` contains the decision `(b && c)`. The short-circuit evaluation of ISO *C* states that the second argument of the operators `&&` and `||` is not evaluated if the result of the operator is already determined by the first argument. The correct identification of hidden control flow is important, for example, to analyze decision coverage. See [3] for further details with respect to code coverage.

Condition is a Boolean expression. We consider only lowest-level conditions, i.e., conditions that do not contain operators with Boolean arguments [8]. A condition is composed of one or more basic blocks. We denote the set of conditions of a decision d as $\mathrm{C}(d)$. The set of all conditions within a program P_i is denoted as $\mathrm{C}(P_i)$.

Input data $\mathbb{ID}$ defines the set of all possible valuations[1] of the input variables of a program.

Test data $\mathbb{TD}$ defines the set of valuations of the input variables that have been generated with structural code coverage analysis done at source-code level. Since exhaustive testing is intractable in practice, $\mathbb{TD}$ is assumed as a true subset of the program's input data space $\mathbb{ID}$: $\mathbb{TD} \subset \mathbb{ID}$. If we would aim for exhaustive testing ($\mathbb{TD} = \mathbb{ID}$) there would be no challenge of structural code-coverage preservation.

Note that a test case consists, besides the test data, also of the expected output behavior of the program. Since we are primarily concerned with the preservation of structural code coverage with consider only the test data.

Reachability valuation $IV_R(x)$ defines the set of valuations of the input variables that trigger the execution of expression x, where x can be a condition, decision, or a basic block.

Satisfiability valuation $IV_T(x)$**,** $IV_F(x)$ defines the sets of valuations of the input variables that trigger the execution of the condition/decision x with a certain result of x: $IV_T(x)$ is the input-data set, where x evaluates to TRUE and $IV_F(x)$ is the set, where x evaluates to FALSE. The following properties always hold for $IV_T(x)$, $IV_F(x)$:

$$IV_T(x) \cap IV_F(x) \supseteq \emptyset$$
$$IV_T(x) \cup IV_F(x) = IV_R(x)$$

Consider the following example of C code to get an intuition about the meaning of the satisfiability valuations:

```
void f (int a,b) {
   if (a==3 && b==2)
      return 1;
   return 0;
}
```

For this code fragment we assume

$$IV_R(\texttt{a==3}) = \{\langle a, b\rangle \mid a, b \in \texttt{int}\}$$

From this assumption it follows that

$$IV_R(\texttt{b==2}) = \{\langle 3, b\rangle \mid b \in \texttt{int}\}$$

(and not the larger set $\{\langle a, b\rangle \mid a, b \in \texttt{int}\}$ due to the hidden control flow caused by the short-circuit evaluation of ISO *C* [3]). It follows that

$$IV_T(\texttt{b==2}) = \{\langle 3, 2\rangle\}$$

[1] Valuation of a variable means the assignment of concrete values to it. The valuation of an expression means the assignment of concrete values to all variables within the expression.

Only those input data that trigger the execution of condition `b==2` and evaluate it to TRUE are within $IV_T(\texttt{b==2})$. With $\langle 3, 2\rangle$ the conditions `a==3` and `b==2` are both executed and evaluated to TRUE. Further, it holds that

$$IV_F(\texttt{b==2}) = \{\langle 3, b\rangle \mid b \in \texttt{int} \wedge b \neq 2\}$$

The definitions of $IV_R(x)$, $IV_T(x)$, and $IV_F(x)$ depend on whether the programming language has hidden control flow, for example, the short-circuit evaluation of ISO *C* [9].

2.2 Statement Coverage (SC)

Statement coverage (SC) requires that every statement of a program P is executed at least once. Statement coverage alone is quite weak for functional testing [10] and should best be considered as a minimal requirement. Using above definitions, we can formally define SC as follows:

$$\forall b{\in}\mathrm{B}(P).\ (\mathbb{TD} \cap IV_R(b)) \neq \emptyset \tag{1}$$

Note that the boundary recognition of basic blocks $\mathrm{B}(P)$ can be tricky due to hidden control-flow. A statement in a high-level language like ISO *C* can consist of more than one basic block. For example, the ISO *C* statement `f=(a==3 && b==2);` consists of multiple basic blocks due to the short-circuit evaluation order of ISO *C* expressions.

Remark 1. Source-line coverage is sometimes used as an alternative to SC in lack of adequate testing tools. However, without the use of strict coding guidelines, source-line coverage is not a serious testing metrics, as it is possible to write whole programs of arbitrary size within one source line.

2.3 Condition Coverage (CC)

Condition coverage (CC) requires that each *condition* of the program has been tested at least once with each possible outcome. It is important to mention that CC does *not* imply DC. A formal definition of CC is given in Equation 2.

$$\forall c{\in}\mathrm{C}(P).\ (IV_T(c) \cap \mathbb{TD}) \neq \emptyset \wedge (IV_F(c) \cap \mathbb{TD}) \neq \emptyset \tag{2}$$

Remark 2. Above definition of CC requires in case of short-circuit operators that each condition is really executed. This is due the semantics of $IV_T()$,$IV_F()$. However, often other definitions are used that do not explicitly consider short-circuit operators (as, for example in [11]), thus having in case of short-circuit operators only a “virtual” coverage since they do not guarantee that the short-circuit condition is really executed for the evaluation to TRUE as well as for the evaluation to FALSE.

2.4 Decision Coverage (DC)

Decision coverage (DC) requires that each *decision* of a program P has been tested at least once with each possible outcome. Decision coverage is also known as *branch*

coverage or *edge coverage*.

$$\forall d \in \mathrm{D}(P).\ (IV_T(d) \cap \mathbb{TD}) \neq \emptyset \wedge (IV_F(d) \cap \mathbb{TD}) \neq \emptyset \tag{3}$$

3 Preservation of Structural Code Coverage

The challenge of structural code-coverage preservation is to ensure for a given structural code coverage of a program P_1 that this code coverage is preserved while the program P_1 is transformed into another program P_2. This scenario is shown in Figure 2. Of course if a program will be transformed, also the sets of basic blocks B or the set of program decisions D may get changed. As shown in Figure 2, the interesting question is whether a concrete code transformation preserves the structural code coverage of interest.

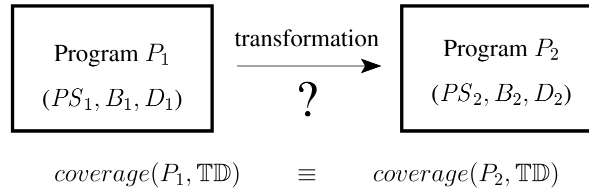


Fig. 2. Coverage-Preserving Program Transformation

When transforming a program, we are interested in the program properties that must be maintained by the code transformation such that a structural code coverage of the original program by the test-data set $\mathbb{TD}$ is preserved to the transformed program. Based on these properties one can adjust a source-to-source transformer or a compiler to use only those optimizations that preserve the intended structural code coverage. These coverage-preservation properties to be maintained have to ensure that whenever the code coverage is fulfilled at the original program by some test data $\mathbb{TD}$ then this coverage is also fulfilled at the transformed program with the same test data:

$$\forall \mathbb{TD}.\ coverage(P_1, \mathbb{TD}) \implies coverage(P_2, \mathbb{TD}) \tag{4}$$

In the following we present several coverage preservation criteria taken from [3]. We use these coverage preservation criteria together with abstract descriptions of the code transformations for the calculation of the coverage profiles.

3.1 Preserving Statement Coverage (SC)

Equation 5 of Theorem 31 provides a coverage preservation criterion for statement coverage. Equation 5 essentially says that for each basic block b' of the transformed program there exists a basic block b of the original program such that reaching b with a given test vector implies that also b' is reached with the same test vector.

Theorem 31 (Preservation of SC) *Assuming that a set of test data $\mathbb{TD}$ achieves* statement coverage *on a given program P_1, then Equation 5 provides a sufficient - and without further knowledge about the program and the test data (there is now knowledge about the test data or the program assumed), also necessary - criterion for guaranteeing preservation of statement coverage on a transformed program P_2. (Proof given in [3])*

$$\forall b' \in \mathrm{B}(P_2)\ \exists b \in \mathrm{B}(P_1).\ IV_R(b') \supseteq IV_R(b) \tag{5}$$

3.2 Preserving Condition Coverage (CC)

To define a coverage preservation criterion for CC (Theorem 32) we use the auxiliary predicate $touches_ID(x, ID)$ given in Equation 6.

The predicate $touches_ID(x, ID)$ is only TRUE if the set of input data ID includes at least the true-satisfiability valuation $IV_T(x)$ or the false-satisfiability valuation $IV_F(x)$ of expression x, where x is either a condition or a decision. The predicate $touches_ID(x, ID)$ is used for the coverage preservation criterion of CC (and also DC) to test whether the evaluation of any expression x of the original program to both, TRUE and FALSE, implies that the test data include at least one element of ID, needed for the coverage of an expression in the transformed program.

$$touches_ID(x, ID) \quad \Rightarrow \quad (IV_T(x) \subseteq ID)\ \vee\ (IV_F(x) \subseteq ID); \tag{6}$$

Equation 7 states that for each condition c' of the transformed program there exists at least one condition of the original program whose coverage implies that c' evaluates to TRUE and there exists at least one condition of the original program whose coverage implies that c' evaluates to FALSE.

Theorem 32 (Preservation of CC) *Assuming that a set of test data $\mathbb{TD}$ achieves* condition coverage *on a given program P_1, then Equation 7 provides a sufficient - and without further knowledge about the program and the test data, also necessary - criterion for guaranteeing preservation of condition coverage on a transformed program P_2. (Proof given in [3])*

$$\begin{aligned}\forall c' \in \mathrm{C}(P_2).\ \exists c \in \mathrm{C}(P_1).\ & touches_ID(c, IV_T(c'))\ \wedge \\ \exists c \in \mathrm{C}(P_1).\ & touches_ID(c, IV_F(c'))\end{aligned} \tag{7}$$

3.3 Preserving Decision Coverage (DC)

To define a coverage preservation criterion for DC (Theorem 33) we use the auxiliary predicate $touches_ID(x, ID)$ given in Equation 6, which is also used for preserving CC.

Equation 8 of Theorem 33 provides a coverage preservation criterion for decision coverage. Equation 8 essentially says that for each decision d' of the transformed program there exists at least one decision of the original program whose coverage implies that d' evaluates to TRUE and there exists at least one decision of the original program whose coverage implies that d' evaluates to FALSE.

Theorem 33 (Preservation of DC) *Assuming that a set of test data* $\mathbb{TD}$ *achieves* decision coverage *on a given program* P_1*, then Equation 8 provides a sufficient - and without further knowledge about the program and the test data, also necessary - criterion for guaranteeing preservation of decision coverage on a transformed program* P_2*. (Proof given in [3])*

$$\forall d' \in \mathrm{D}(P_2).\ \exists d \in \mathrm{D}(P_1).\ \mathit{touches_ID}(d, IV_T(d'))\ \wedge \\ \exists d \in \mathrm{D}(P_1).\ \mathit{touches_ID}(d, IV_F(d')) \tag{8}$$

4 Automatic Calculation of Compilation Profiles

This section discusses the concepts and implementation behind automatic calculation of coverage profiles.

4.1 Program Model

For modeling control flow, the sequence of execution is defined by a set of labeled CFG edges $R : E \times \Lambda \times \Delta$, where $E : N \times N$ are the CFG edges with $N : B \cup C \cup \{s, t\}$, $\Lambda : \{T, F\} \times \{T, F, X\}$, and $\Delta : \{\delta_1, \ldots, \delta_{|R|}\}$. The special labels Λ and Δ are used to include information about control flow that depends on condition/decision evaluations and influence of input valuations.

Condition/decision labels $\ell \in \Lambda$ are used in case of condition nodes to determine the path a program uses when the control flow forks depending on the result of a condition evaluation. For flexibility in assigning condition results to different decision outcomes the condition/decision labels are two-parted. The first part defines the condition evaluation result using the symbols T and F for *true* and *false*. The second part of the label determines the decision result correlated with the condition result accumulated so far. It can be T, F or X if the decision outcome is not yet determined. Note that X is only allowed for edges originating and destinating inside the *decision hypernode*. All outgoing edges of a decision must carry a unique decision-label with T or F.

Each edge e_i in the graph is assigned a *valuation set* $\delta_i \in \Delta$. This valuation set represents all the valuations of the program input that trigger the execution of a path going through edge e_i. For each node v, except s and t, we have a continuity relation of the form

$$\bigcup_{e_i \in IN(v)} \delta_i = \bigcup_{e_j \in OUT(v)} \delta_j \tag{9}$$

where $IN(v)$ denotes the incoming edges of v and $OUT(v)$ the outgoing edges of v. In other words, execution paths entering a node must leave the node at least on one

outgoing edge. The only exceptions are the entry-node s being the source and the exit-node t being the sink of each execution path.

4.2 Analyzing Code Optimizations

For analyzing the effect of code optimizations we model the valuation relations between the original and the transformed code. Based on the continuity relation (Equation 9) it is easy to obtain simple subset relations ($\subseteq$) between the valuation sets on incoming and outgoing edges inside each program graph. This can be done by walking through each node of the CFG and applying the continuity relation in forward and backward direction. These subset relations are the basic input for coverage preservation analysis.

A code transformation adds additional relations of valuation sets between the original and the transformed code, characterizing how the transformation forms the valuation sets of the edges in the transformed code based on the valuation sets of the edges in the original code. These relations can be propagated along the CFG based on the transitivity of subset relations.

4.3 The Mathematica Implementation

The implementation of the coverage-profile calculation was done using *Mathematica*, a fully integrated environment for technical and mathematical computing [12].

In a preparation-phase the control-flow graphs with the node sets B and C, the decision set D and the edge set R must be converted to the internal data structures of the program system. Each edge e in R is implemented as a tuple $\langle v, w, \ell, \delta_i \rangle$ where v is the start- and w is the end-point of the edge. ℓ is the two-parted condition/decision label as described above (or empty if the edge is not originating at a condition node) and δ is a unique identifier for the valuation set.

Reachability and satisfyability valuation are reproduced internally by collecting the valuation sets on incoming and outgoing edges. $IV_R(x)$ is calculated as abstract union of all valuation sets on the incoming edges of node x. To calculate $IV_T(x)$ the union of all outgoing edges of x labelled with T are calculated and for $IV_F(x)$ all edges with label F are mentioned. Dependent wether x is a condition or a decision, the information is extracted from the condition or decision label.

We construct an auxiliary graph (derived from the CFG) for maintaining the equality relations ($=$) or subset relations ($\subseteq$) between valuation sets. The nodes of the auxiliary graph represent valuation sets or unions of valuation sets. A directed edge $\delta_i \rightarrow \delta_j$ is included in the support graph iff $\delta_i \supseteq \delta_j$ is true. In case of $\delta_i = \delta_j$ the auxiliary graph contains edges between δ_i and δ_j in both directions.

After constructing auxiliary graphs for the original code as well as the transformed code, these graphs are glued together by adding the additional relations caused by the code transformation. These subset relations form the abstract description of the code transformation that we use for the calculation of the coverage profiles. Creating a graph-reconstruction language [13, 14] that records the transformation relation while reconstructing the CFG is a possible extension for the future.

So far we have implemented preservation analyses for *statement coverage* (SC), *condition coverage* (CC), and *decision coverage* (DC). They get descriptions of the

original CFG and the transformed CFG. Beside documentary information they output a verdict *true* or *false* about the ability of the transformation to preserve the mentioned coverage. The correctness of this verdicts relies on providing a correct and precise abstract description of the code transformation.

5 Examples of Analyzing Coverage-Preservation

This sections shows the coverage preservation analysis for several code optimizations. To avoid confusion when relating the valuation sets of the original code and the transformed code, we denote δ_i the valuation sets of the original code and ϱ_i the valuation sets of the transformed code. The results on coverage preservation are summarized in Table 1.

5.1 Condition Reordering with Short-Circuit Evaluation

Algebraic simplifications use algebraic properties of operators like associativity, commutativity and distributivity to simplify expressions [1]. Although these simplifications produce logically equivalent expressions, they may cause unexpected changes in the flow of control. Under certain circumstances these changes can disrupt structural code coverage if they change the order of conditions. This is demonstrated in the following example of a branch with short-circuit evaluation.

The case study demonstrates condition reordering in an if-statement with two conditions connected by a logical AND operator with short-circuit evaluation. In a programming language the program code and the optimized code could look similar as in the following C-styled example:

```
if ( A && B )                          if ( B && A )
   thenBlock                              thenBlock
else              changed to           else
   elseBlock                              elseBlock
```

Additionally, short-circuit evaluation of conditions is assumed, a technique used in several programming languages. In C/C++, e.g., logical expressions inside an if-statement are evaluated from left to right. If evaluation of further terms could not change the result anymore, evaluation stops and the branch is executed immediately. In the example, the second condition is not evaluated, if the result of the first condition evaluates to *false*.

Figure 3 shows the internal graph models for this use case with the original program on the left side and the transformed program on the right side. As a convention the symbols δ are used to note the valuation sets of the original program and symbols ϱ are used for the transformed program. In the original program, the described short-circuit branch is implemented with the edge from condition A to the else-block 4. In the transformed program the short-circuit branch connects condition B with else-block 14.

Changing the condition order by swapping the conditions will not change the valuation-sets of the decision result. This is denoted by the equality relations ($=$) between δ_6, ϱ_6 and δ_7, ϱ_7. Therefore, applying the preservation condition for DC (Equation 8) and for SC (Equation 5) will give a positive preservation verdict. But the distribution of the valuation sets of the conditions inside the decision are changed. Condition

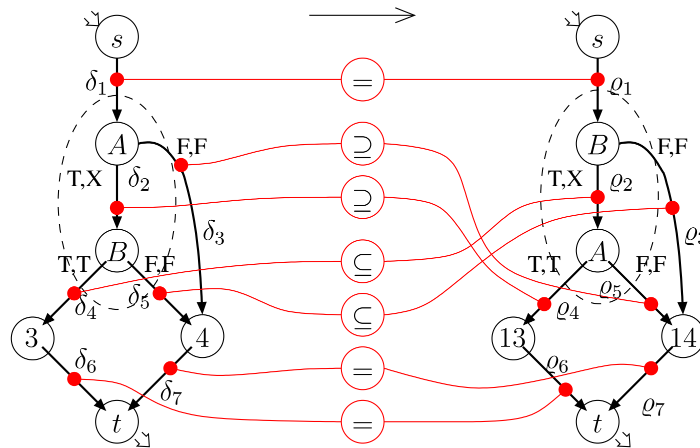


Fig. 3. Transformation Relation for Condition Reordering (with short-circuit evaluation)

B in the transformed code snippet will now decide on a bigger valuation set than in the untransformed program while condition A in the transformed program decides on a subset of the possible valuations. Applying the preservation condition for CC (Equation 7) therefore results in a negative preservation verdict.

The sample output of the implemented analyzing function in Figure 4 shows how the function makes use of the preservation criteria to show, that statement coverage is preserved. The tool walks through each statement node of the transformed code. Using the continuity relation together with the additional subset relations on the valuation sets it determines those valuation sets which are a subset of valuation set $IV_R(x)$ of the currently investigated node x. Finally, it searches for a node in the original code with a valuation-set that is member of the related valuation-sets. In the first case this happens with node 3 and it's valuation set $IV_R(3) = \delta_4$. The same principle is used to find node 4 as a counterpart for node 14.

The last line of the listing gives as the function result the final decision, which is *true* in this case. This result can be used to be included into a coverage profile.

5.2 Loop Peeling

The transformation called *loop peeling* replaces the first k iterations from the beginning of a loop and inserts k copies of the body together with increment and test code of the loop index variable immediately ahead of the loop [1]. A simplified example of this optimization is shown in Figure 5, where the compiler has peeled out the first iteration of the loop, placing one copy of the loop body and the loop termination test in front of the loop.

From point of view of code coverage analysis, this little change in code structure has severe effects on preservation of all coverage criteria. In the original program SC, CC, and DC can be achieved by executing one iteration of the loop. After application of the transformation, the same test data will not enter the loop, because the first iteration has been executed in advance.

** SC-Preservation **
B(P2): $\{13, 14\}$
B(P1): $\{3, 4\}$
$\rightarrow\rightarrow\rightarrow 1$
IV$_R$(13)== $\{\rho_4\}$ of P2 is related with
$\{\{\delta_4\}, \{\delta_6\}, \{\rho_4\}, \{\rho_6\}\}$
Nodes of P1 satisfying preservation condition: $\{3\}$
Accumulated scpf: True
$\rightarrow\rightarrow\rightarrow 2$
IV$_R$(14)== $\{\rho_3, \rho_5\}$ of P2 is related with
$\{\{\delta_3\}, \{\delta_5\}, \{\delta_7\}, \{\rho_3\}, \{\rho_5\}, \{\rho_7\}, \{\delta_3, \delta_5\}, \{\rho_3, \rho_5\}\}$
Nodes of P1 satisfying preservation condition: $\{4\}$
Accumulated scpf: True
True

Fig. 4. Sample Output Analyzing Statement Coverage for a IF-Statement with Two Conditions (with short-circuit evaluation)

Formal analysis cannot prove coverage preservation, because the body of the loop is only triggered by a subset ϱ_4 of the original valuation subset δ_2. Therefore SC will fail for b' = "B", because no statement b in the original program could be found such that $IV_R(b') \supseteq IV_R(b)$. Proofing preservation of CC and DC fails for similar reasons.

5.3 Loop Inversion

Loop Inversion, in source-language terms, transforms a `while` loop into a `do-while` loop [1]. The loop closing test is moved from the beginning of the loop to the end of the loop. In the simplest case this requires, that it is save to execute the loop body at least once. Otherwise, a test has to be generated in front of the loop to check the exit condition. This latter case is illustrated in Figure 6.

Although the relation of the valuation sets between the original and the transformed code contains many equalities, only statement coverage is preserved. This is, because the moved loop closing decision in the transformed program only decides on a subset of the input valuations compared with the original program, which is expressed by the union operation ($\cup$) on the right side of the equality relation. This relation is induced by the subset-relation between ϱ_1 and ϱ_2.

5.4 Condition Reordering without Short-Circuit Evaluation

This example goes back to the condition reordering example presented in Section 5.1. The example presented in this subsection is a variation where all conditions are executed independently of the outcome of the other conditions of the decision. Besides SC and DC, also CC is now preserved. The main difference here is, that each condition decides on the full valuation-set $\delta_2 \cup \delta_3 = \varrho_2 \cup \varrho_3$, although the distribution between δ_2, δ_3 on one side and ϱ_2, ϱ_3 on the other side may differ.

The CFG in Figure 7 also shows an application for the two-parted condition/decision label. Although condition "B" in the original code on the left side can

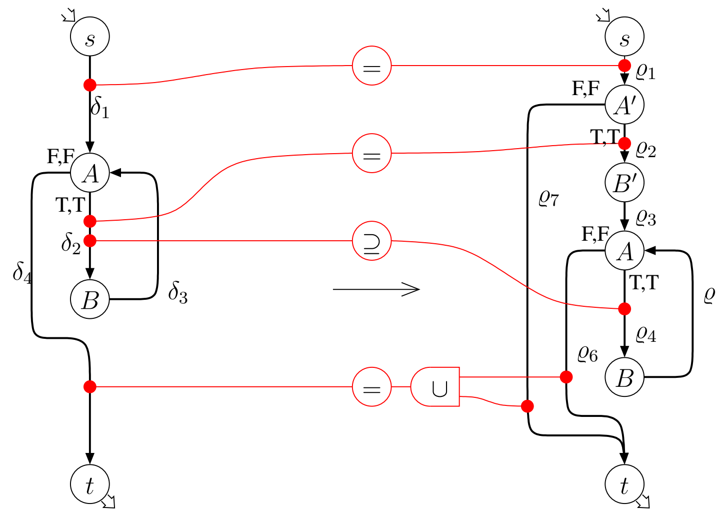


Fig. 5. Transformation Relation for Loop Peeling

decide independently of the result of condition “A” for *true*, the decision result must be *false* if the result of evaluating condition “A” was *false*. The same is true in the transformed program when condition “A” decides *true* but the result of condition “B” was *false*.

Code Optimization	*Coverage Preservation*		
	SC	*CC*	*DC*
Cond. reordering (without short-circuit)	✓	·	✓
Cond. reordering (with short-circuit)	✓	✓	✓
Loop peeling	·	·	·
Loop inversion	✓	·	·

Table 1. Calculated Coverage Profiles

6 Summary and Conclusion

In this paper we addressed the rather novel field of preserving structural code coverage during program transformation. A code transformer that take care of preserving structural code coverage has many interesting applications. For example, this allows the realization of reliable and portable test-data generators. Besides functional software testing, this is even interesting for measurement-based timing analysis.

Our approach is based on the calculation of so-called coverage profiles, which are tables that store the information of what code transformations guarantees the preservation of which structural code-coverage metric. To calculate these coverage profiles, we developed a formal coverage preservation criteria for each structural coverage metric

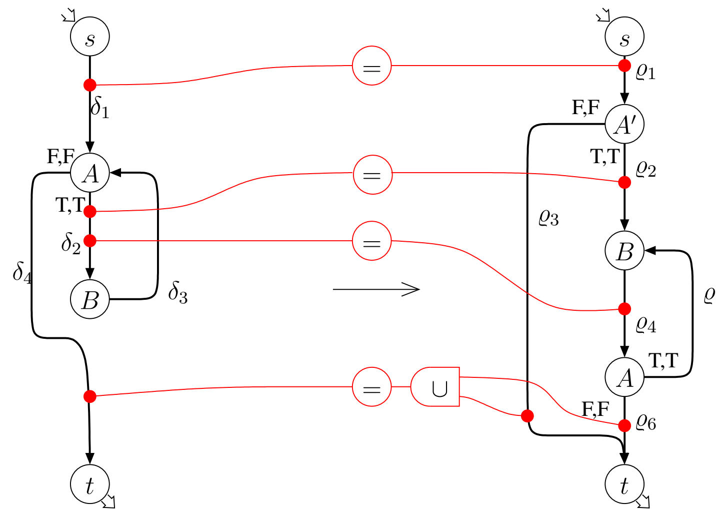


Fig. 6. Transformation Relation for Loop Inversion

and infer it with the abstract descriptions of the code transformations. We have calculated such coverage profiles for statement coverage (SC), condition coverage (CC), and decision coverage (DC).

As future work, we are focusing on extending the calculation of coverage profiles to more complex structural code-coverage metrics like the modified condition-decision criterion (MCDC) or a scoped path coverage.

References

1. Muchnick, S.S.: Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers, Inc. (1997) ISBN 1-55860-320-4.
2. Kirner, R.: SCCP/x - a compilation profile to support testing and verification of optimized code. In: Proc. ACM Int. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'07), Salzburg, Austria (2007) 38–42
3. Kirner, R.: Towards preserving model coverage and structural code coverage. EURASIP Journal on Embedded Systems **2009** (2009) doi:10.1155/2009/127945.
4. Wenzel, I., Kirner, R., Rieder, B., Puschner, P.: Measurement-based timing analysis. In: Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Porto Sani, Greece (2008)
5. Kirner, R., Puschner, P., Wenzel, I.: Measurement-based worst-case execution time analysis using automatic test-data generation. In: Proc. 4th International Workshop on Worst-Case Execution Time Analysis, Catania, Italy (2004) 67–70
6. Vilkomir, S.A., Bowen, J.P.: Formalization of software testing criteria using the z notation. In: Proc. 25th Annual International Computer Software and Applications Conference, Honolulu, Hawaii, USA (2001) 351
7. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison-Wesley (1997) ISBN 0-201-10088-6.
8. Chilenski, J.J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical Report DOT/FAA/AR-01/18, Boeing Commercial Airplane Group (2001)

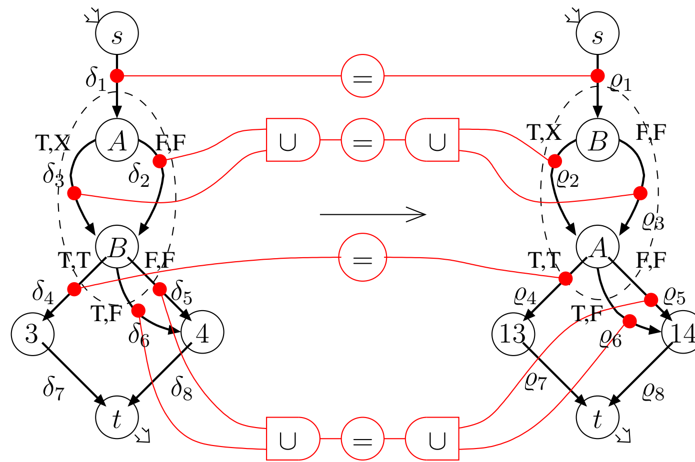


Fig. 7. Transformation Relation for Condition Reordering (without short-circuit evaluation)

9. ISO: Programming Languages – *C*. 2nd edn. ISO/IEC 9899:1999 (1999) Technical Committe: JTC 1/SC 22/WG 14.
10. Myers, G.J.: The Art of Software Testing. John Wiley & Sons (1979)
11. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutoral on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, National Aeronautics and Space Administration, Hampton, Virginia (2001) available in pdf format.
12. Wolfram, S.: The Mathematica Book, 4th ed. Cambridge University Press (1999)
13. Lacey, D., Jones, N.D., Wyk, E.V., Frederiksen, C.C.: Proving correctness of compiler optimizations by temporal logic. SIGPLAN Not. **37** (2002) 283–294
14. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, ACM Press (2003) 220–231

On Undecidability Results of Real Programming Languages

Raimund Kirner[1], Wolf Zimmermann[2], and Dirk Richter[2]

[1] Vienna University of Technology, Real Time Systems Group, A-1040 Wien, Treitlstr. 3/182-1,raimund@vmars.tuwien.ac.at
[2] Martin-Luther Universitt Halle-Wittenberg, Institut fr Informatik, 06099 Halle/Saale, Germany, zimmer@informatik.uni-halle.de

Abstract. Often, it is argued that some problems in data-flow analysis such as e.g. worst case execution time analysis are undecidable (because the halting problem is) and therefore only a conservative approximation of the desired information is possible. In this paper, we show that the semantics for some important real programming languages – in particular those used for programming embedded devices – can be modeled as finite state systems or pushdown machines. This implies that the halting problem becomes decidable and therefore invalidates popular arguments for using conservative analysis.

1 Introduction

Many program analysis approaches are conservative, i.e. they only deliver approximate results. They guarantee positive results but may overestimate the negative side. E.g., worst-case execution time (WCET) analysis has to deliver an upper bound of the execution time, but this bound may be an overestimation of the real WCET [1, 2]. An other example is live-variable analysis (variables at a certain program point that contain values which are needed for further computations). The set of live variables may be overestimated. However for any variable not in this set, it is guaranteed that its value will never be read. The popular argument is that to derive these informations is not computable in the sense of a Turing-Machine. However, undecidability results or computational intractability results such e.g. the halting problem are based on the assumption that every variable contains an integer (or may store values of an infinite range). A closer look to language definitions such as, e.g., *C*, *C++*, *Java*, *Fortran*, *Ada* or *C#* shows that the base types have finite range. In languages such as *C* or *C++* even pointers or references are of finite range. Most compilers map pointers and references to addresses of the target processor, i.e. they are also finite range. Moreover in programming language (restrictions) for embedded devices there are either no pointers or it is explicitly stated that anonymous objects are allocated statically. This also implies that variables containing pointers can only store values of finite range.

Our aim is to show that restricting variables to finite data types leads in many cases (e.g. *C* or *Java*, but not *Ada*) to the decidability of the halting

problem and allows exact computation of programming analysis information such as those mentionend above. It is not claimed that it is practically feasible to compute exact program analysis informations, but the argument not to do it cannot be based on undecidability results.

Our approach is as follows, we show for a representative subset of ISO *C* that its semantics as a state transition system is a pushdown automaton. Since reachability, LTL or CTL model checking on pushdown automata is decidable, the halting problem of this subset becomes decidable. Any analysis that can be expressed as a formula in LTL or CTL becomes decidable and can therefore be computed exactly. In general the semantics of the whole *C*-language can be formalized in this way but this would go beyond the limitations of such an article.

In programming languages there are only two reasons for being infinite state: either some variables may store values of an infinite type (as e.g. references for an unlimited memory) or there is no limitation on the recursion depth. Note that an unlimited heap allocation would require infinite reference types. If each variable can only store a finite range of values, any limitation on recursion depth would lead to a *finite* number of states and therefore to a finite state system. Since programming languages for embedded systems often restrict recursion depth and allows only finite types, the semantics of programs in these languages are finite state machines. Interestingly, for some programming languages the halting problem even becomes undecidable if all variables only can store Boolean values (it is possible to simulate a Turing Machine on the run time stack, when reference parameters, recursion, procedure parameters, local procedures and global/local variables are allowed).

2 Pushdown Systems

A *pushdown system* is a tuple $\Pi \triangleq (\Sigma, \Gamma, I, \rightarrow)$ where Σ is a finite set of *states*, Γ is a finite set (the *stack alphabet*), $I \subseteq \Sigma \times \Gamma^*$ is the set of *initial configurations*, and $\rightarrow \subseteq \Sigma \times \Gamma \times \Sigma \times \Gamma^*$ is a finite relation (the *transition rules*). As usual X^* denotes all finite sequences of elements of a set X, ε the empty sequence, xy denotes the concatenation of two sequences $x, y \in X^*$, and $|x|$ the length of the sequence x. An element $(\sigma, \gamma) \in \Sigma \times \Gamma^*$ is called a *configuration* of Π. Let $\kappa \triangleq (\sigma, \gamma\bar{\gamma})$ be a configuration where $\gamma \in \Gamma$. Then the *head of the configuration* κ is the pair $hd(\kappa) \triangleq (\sigma, \gamma) \in \Sigma \times \Gamma$. A *direct derivation* $\Rightarrow_\Pi \subseteq \Sigma \times \Gamma^* \times \Sigma \times \Gamma^*$ is defined by the following inference rule: $\dfrac{(\sigma, \gamma) \rightarrow (\sigma, \gamma')}{(\sigma, \gamma\bar{\gamma}) \Rightarrow (\sigma', \gamma'\bar{\gamma})}$ for all $\bar{\gamma} \in \Gamma^*$.

The *derivation relation defined by* Π is the reflexive transitive closure $\overset{*}{\Rightarrow}_\Pi$ of $\Rightarrow_\Pi$. A configuration (σ, γ) is *final in* Π iff there is no configuration (σ', γ') such that $(\sigma, \gamma) \Rightarrow_\Pi (\sigma', \gamma')$. A configuration κ is *reachable* in Π if there is an initial configuration $\kappa_0 \in I$ such that $\kappa_0 \overset{*}{\Rightarrow}_\Pi \kappa$. The set of reachable configurations of Π is denoted by $Post^*(\Pi)$.

A *run of* Π is a finite or infinite sequence $\kappa \triangleq \kappa_0\kappa_1 \cdots$ of configurations such that $\kappa_0 \in I$ and $\kappa_i \Rightarrow_\Pi \kappa_{i+1}$ for all $i \in \mathbb{N}, i + 1 < |\kappa|$. If κ is finite, the last

configuration must be final. It is possible to transform finite runs to infinite runs by adding the transition rules $hd(\kappa) \rightarrow hd(\kappa)$ for all final configurations κ.

3 Deciding the halting problem on *C*

We now show how a reasonable subset of *C* can be mapped to symbolic pushdown systems (thereby defining the semantics of this subset). In contrast to *C*, we do not limit recursion depth in this dialect. Such a limitation would lead to a finite state system, which is a special case of pushdown systems (i.e. the results implied by the assumptions of this section are more general).

Remark 1. In *C* recursion depth is implicitly limited because it is possible to obtain addresses from local variables using the `&`-operator and the language definition of ISO *C* [3] states that addresses are of finite range (depending on the target). Thus, only finitely many local variables can be stored which implicitly limits the depth of the runtime stack.

```
⟨prog⟩     ::= ⟨decl⟩+
⟨decl⟩     ::= ⟨vardecl⟩ | ⟨procdecl⟩
⟨vardecl⟩  ::= ⟨type⟩ identifier [=⟨expr⟩] ;
⟨procdecl⟩ ::= type identifier ([⟨pars⟩]) ⟨block⟩
⟨type⟩     ::= int | void | ⟨type⟩ *
⟨pars⟩     ::= (⟨par⟩ ,)* ⟨par⟩
⟨par⟩      ::= ⟨type⟩ identifier
⟨stat⟩     ::= ⟨assign⟩ | ⟨call⟩ | ⟨vardecl⟩ | ⟨if⟩ | ⟨while⟩ | ⟨block⟩ | ⟨return⟩
⟨assign⟩   ::= ⟨des⟩=⟨expr⟩;
⟨call⟩     ::= ⟨name⟩([args]) ;
⟨args⟩     ::= ⟨expr⟩(,⟨expr⟩)*
⟨if⟩       ::= if (⟨expr⟩)⟨stat⟩[else⟨stat⟩]
⟨while⟩    ::= while (⟨expr⟩)⟨stat⟩
⟨block⟩    ::= {⟨stat⟩*}
⟨return⟩   ::= return [⟨expr⟩] ;
⟨expr⟩     ::= ⟨conj⟩(||⟨conj⟩)*⟩
⟨conj⟩     ::= ⟨rel⟩(&&⟨rel⟩)*
⟨rel⟩      ::= ⟨sum⟩[(== | < | <= | != | > | >=)⟨sum⟩]
⟨sum⟩      ::= ⟨term⟩((+ | -)⟨term⟩)*
⟨term⟩     ::= ⟨unexpr⟩((* | / | %)⟨unexpr⟩)*
⟨unexpr⟩   ::= [!|-]⟨factor⟩
⟨factor⟩   ::= ⟨des⟩ | ⟨call⟩ | const
⟨des⟩      ::= ** ⟨name⟩
⟨name⟩     ::= identifier
```

Fig. 1. Syntax of a *C-K*

Fig. 1 shows the syntax of *C-K*, a representative subset of *C*. The constructs have the traditional semantics as in *C*. One of the declarations must be a function `main`. We assume that the rules of static semantics are satisfied. Table 1 shows some notions on static informations about the program. The details of these notions are discussed in the following.

GLOB	set of global variable identifiers in a program
PROC	set of global procedure identifiers in a program
LOC_p	set of local variables of procedure $\mathtt{p} \in \mathsf{PROC}$
LABEL_p	set of labels associated with the instructions of procedure $\mathtt{p} \in \mathsf{PROC}$
EXPR_p	set of labels associated with expressions in procedure p.

Table 1. Static Information on Programs

We don't consider arrays because their semantics in *C* is defined by pointers and access to its element is defined by pointer arithmetic. We further do not consider structs or unions because accesses to fields can be directly implemented using pointers. A certain amount of the heap can be reserved using `malloc(int)`. It returns an address on the heap. The heap is finite since the number of addresses is finite. We don't allow the address operator. It is therefore not possible to access local variables of stack frames except for the top stack frame. For simplicity, we don't allow assignment expressions (and use assignment statements instead) and assume a left-to-right evaluation order. *Instructions* are statements (except blocks) and expressions. *Declarations* declare entities or procedures. A declaration is *global* if it is declared in a program, otherwise it is called *local*, i.e., it is contained in a block. Such declarations are uniquely associated with a procedure and are variables. We therefore call a variable declaration *local* to a procedure `p` iff it is declared in the block of `p` (sub-blocks are also allowed). For simplicity, we assume that integer variables and pointer variables store (signed) integers and addresses that can be represented by k Bits. BIT^k denotes the set of bit sequences of length k.

Example 1. Fig. 2 shows a *C-K*-program recursively computing faculties. It holds $\mathsf{GLOB} = \{\mathtt{n}\}$, $\mathsf{PROC} = \{\mathtt{fak}, \mathtt{main}\}$, $\mathsf{LOC}_{\mathtt{fak}} = \{\mathtt{n}\}$, and $\mathsf{LOC}_{\mathtt{main}} = \{\mathtt{x}\}$ For each instruction, the label is added as superscript. With these labels, it is $\mathsf{EXPR}_{\mathtt{fak}} = \{6, 7, 8, 10, 12, 13, 14, 15, 16, 17\}$ and $\mathsf{EXPR}_{\mathtt{main}} = \{0, 2, 3\}$.

```
int n;
int fak(int n) {
  if^9 (n^6<=^8 1^7) return^11 1^10;
  return^18 fak^15(n^12-^14 1^13)*^17 n^16;
}
void main() {
  n=^1 2^0;
  int x=^4 fak^3(n^2);
  return^5;
}
```

Fig. 2. A *C-K*-program

We now show that the semantics of the *C*-subset can be formally defined by a pushdown system $\Pi \triangleq (\Sigma, \Gamma, I, \rightarrow)$. Intuitively, the set Σ of states represents

the global variables and the memory and the stack of the pushdown system represents the procedure stack. The stack alphabet Γ defines the set of possible procedure frames. Table 2 shows the notions used for the formal definition of the pushdown system defining the semantics of a *C*-program. The details are explained in the following paragraphs.

STORE^k_V	set of stores for variables V holding sequences of k Bits
MEM^n_m	set of m-Bit addressed memories holding sequences of n Bits
REG^k_R	set of all register assignments for registers R holding sequences of n Bits
FRAME_p	set of stack frames for procedure p.

Table 2. Notions used for the formal definition of the semantics of *C-K*

Stores are used to model the storage for global and local variables. Memories are used to model heaps. Registers are used to store intermediate values when evaluating expressions. This is needed since functions calls are expressions that may have side-effects. Because of recursion, a runtime stack is needed to maintain the currently active procedure calls. The stack frames are the elements of this runtime stack.

Formally, a *store* for a set V of variables (represented by their names) for storing sequences of k bits is a mapping $\sigma : \mathsf{VAR} \rightarrow \mathsf{BIT}^k$. A store for variables $x_1, \ldots, x_n$ with values $v_1, \ldots, v_n$ is denoted by $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$. The memory is a mapping $mem : \mathsf{BIT}^k \rightarrow \mathsf{BIT}^k$. Thus, $\Sigma \triangleq \mathsf{STORE}^k_{\mathsf{GLOB}} \times \mathsf{MEM}^k_k \cup \{err\}$ where $err \notin \mathsf{STORE}^k_{\mathsf{GLOB}} \times \mathsf{MEM}^k_k$ represents a runtime error that causes to terminate exceptionally the program.

Example 2. For the set GLOB in Example 1 $\{\mathtt{n} \mapsto 5\}$ is a store where the value 5 is stored at variable `n`. $\{00 \mapsto 0, 01 \mapsto 1, 10 \mapsto 2, 11 \mapsto 3\}$ is an example of a 2-Bit addressed memory storing at an address a the value a.

Lemma 1. $|\Sigma| = 2^{|\mathsf{GLOB}| \cdot k + 2^k} + 1$

Thus, the number of states is finite.

Intuitively, a stack frame consists of the instruction to be executed next, a store for the local variables, and a store for registers (required for evaluation of expressions). A function result is also stored in a register. For identifying the instructions of a program each instruction is associated with a unique label. In particular, the sets of labels of the procedures are pairwise disjoint. A register assignment of a procedure p is a mapping $\rho : \mathsf{EXPR}_p \rightarrow \mathsf{BIT}^k$. A register assignment that assigns values $u_1, \ldots, u_k$ to registers $r_1, \ldots, r_k$ is denoted by $\begin{array}{c|c|c} r_1 & \cdots & r_k \\ \hline u_1 & \cdots & u_k \end{array}$ Thus, a *stack frame for procedure* $\mathtt{p}$ is a tuple $(l, \sigma, \rho) \in \mathsf{LABEL}_p \times \mathsf{STORE}^k_{\mathsf{LOC}_p} \times \mathsf{REG}^k_{\mathsf{EXPR}_p} \triangleq \mathsf{FRAME}_p$. Then, the stack alphabet can be defined as

$$\Gamma \triangleq \biguplus_{p \in \mathsf{PROC}} \mathsf{FRAME}_p$$

Example 3. $\left(6, \{n \mapsto 2\}, \frac{6|7|8|10|12|13|14|15|16|17}{4|0|2|9|-1|22|47|88|2|9}\right)$ denotes a stack frame for procedure `fak` in Fig. 2.

Lemma 2. $|\mathsf{FRAME}_p| = |\mathsf{LABEL}_p| \cdot 2^{k \cdot (|\mathsf{LOC}_p| + |\mathsf{EXPR}_p|)}$ *and* $|\Gamma| = \sum_{p \in \mathsf{PROC}} |\mathsf{FRAME}_p|$

Thus, the stack alphabet is also finite.

$first_p$	label of the first instruction of a procedure p
$next_l$	label of instruction being executed after instruction labelled l
yes_l	label of the first instruction in the positive case of a conditional or loop
no_l	label of the first instruction in the negative case of a conditional or loop
$odp_l(i)$	label of i-th operand
$par_p(i)$	i-th parameter of procedure p

Table 3. Control-flow and Data-flow in *C-K*

Each procedure has a uniquely defined first instruction. Assignments, procedure calls, expressions, program have a unique label, and (except for loops, conditionals, and calls, and return statements) have a unique next instruction. The instruction executed after checking a condition of an if- or while-statement depends on the outcome of the decision. Therefore, there are two possibilities for the next instruction. Expressions, if- and while-statements need the values of their operands and conditional expressions, respectively. Table 3 shows the corresponding functions for the control- and data-flow. Note that these functions are statically defined for each program.

Example 4. In Fig. 2, the instruction with label 6 is the first instruction of procedure `fak` and the instruction with label 0 is the first instruction of procedure `main`. Thus, $first_{\texttt{fak}} = 6$ and $first_{\texttt{main}} = 0$. The instruction being executed after the instruction at label 0 is the instruction at label 1. Thus, $next_0 = 1$. Fig. 4(a) shows the complete control-flow for the program in Fig. 2. Not that the instruction to be executed after label 9 is the instruction at label 10 if the condition evaluates to a non-zero value and the instruction at label 12 if the condition evaluates to zero. Thus $yes_9 = 10$ and $no_9 = 12$. Fig. 4(b) shows the data-flow (as a use-def-chain) for the program in Fig. 2. E.g. the operands of the instruction at label 14 are at labels 12 and 13. Thus $opd_{14}(1) = 12$ and $opd_{14}(2) = 13$. The operand of the instruction at label 9 is at label 8, i.e. $opd_9(1) = 8$. The operand of the return-instruction at label 11 is at label 10, i.e. $opd_{11}(1) = 10$.

Table 3 shows the transition rules of the program. It uses the notations shown in Table 4. In the initial state, all global variables are initialized with 0, the contents of the memory is uninitialized, the first procedure to be executed is `main`, and the first instruction being executed is the first instruction of `main`. All local variables of `main` are uninitialized, i.e. any store for the local variables of `main` is allowed for the first configuration. The same remark applies for any register

$$
\begin{array}{ll}
\mathit{Trans}_s & \triangleq \{start \to ((o,m),(\mathit{first}(\texttt{main}),\sigma,\rho)) : m \in \mathsf{MEM}_k^k, \sigma \in \mathsf{STORE}^k_{\mathsf{LOC}_{\texttt{main}}}, \rho \in \mathsf{REG}^k_{\mathsf{EXPR}_{\texttt{main}}}\} \\
\mathit{Trans}_c & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^c_L))\} \\
\mathit{Trans}_v & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^x_L)) : var(L) \in GLOB, x = \sigma'(var(L))\} \\
 & \cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^x_L)) : var(L) \in LOC_{proc(L)}, x = \sigma'(var(L))\} \\
 & \cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^x_L)) : x = mem(\rho(reg(opd_L(1))))\} \\
\mathit{Trans}_\ominus & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^x_L)) : x = \ominus(\rho(opd_L(1)))\}, \qquad \ominus \in \{\texttt{!},\texttt{-}\} \\
\mathit{Trans}_\oplus & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^x_L)) : x = \ominus(\rho(opd_L(1)),\rho(opd_L(2))) \neq undef\} \\
 & \cup\{((\sigma,m),(L,\sigma',\rho)) \to \kappa : \kappa \in \Sigma \times \Gamma^*, \ominus(\rho(opd_L(1)),\rho(opd_L(1))) = undef\}, \\
 & \qquad \oplus \in \{\texttt{+},\texttt{*},\texttt{-},\texttt{/},\texttt{\%},\texttt{==},\texttt{!=},\texttt{<},\texttt{<=},\texttt{>},\texttt{>=}\} \\
\mathit{Trans}_{\texttt{\&\&}_1} & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(no_L,\sigma',\rho|^0_L)) : \rho(opd_L(1) = 0\} \\
 & \cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(yes_L,\sigma',\rho|^0_L)) : \rho(opd_L(1) \neq 0\} \\
\mathit{Trans}_{\texttt{\&\&}_2} & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^{\rho(opd_L(2))}_L)) : opd_L(1) \neq 0\} \\
 & \cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^1_L)) : opd_L(1) \neq 0\} \\
\mathit{Trans}_{\texttt{||}_1} & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(yes_L,\sigma',\rho|^0_L)) : \rho(opd_L(1) = 0\} \\
 & \cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(no_L,\sigma',\rho|^0_L)) : \rho(opd_L(1) = 0\} \\
\mathit{Trans}_{\texttt{||}_2} & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^{\rho(opd_L(2))}_L)) : opd_L(1) = 0\} \\
 & \cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho|^1_L)) : opd_L(1) \neq 0\} \\
\mathit{Trans}_{\texttt{=}} & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma|^v_x,m),(next_L,\sigma',\rho)) : x \triangleq opd_L(1) \in \mathsf{GLOB}, v \triangleq \rho(opd_L(2))\} \\
 & \cup((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma'|^v_x,\rho)) : x \triangleq opd_L(1) \in \mathsf{LOC}_{proc(L)}, v \triangleq \rho(opd_L(2))\} \\
 & \cup\{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m|^v_{\rho(x)}),(next_L,\sigma',\rho)) : x \triangleq opd_L(1) \notin GLOB \cup LOC_{proc(L)}, v \triangleq \rho(opd_L(2))\} \\
\mathit{Trans}_{\texttt{p(x)}} & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(\mathit{first}_p,\sigma'',\rho')(L,\sigma',\rho)) : \rho \in \mathsf{REG}^k_{\mathsf{EXPR}_p}, p = opd_L(1), \sigma'' \in \mathsf{STORE}^k_{\mathsf{LOC}}, \sigma''(par_p(i)) = \rho(opd_L(i+1))\} \\
\mathit{Trans}_{d_1} & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma',\rho))\} \\
\mathit{Trans}_{d_2} & \triangleq \{((\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(next_L,\sigma'|^v_x,\rho)) : x = opd_L(1), v = \rho(opd_L(2)\} \\
\mathit{Trans}_i & \triangleq \{(\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(yes_L,\sigma',\rho)) : \rho(opd_L(1)) \neq 0\} \cup \{(\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(no_L,\sigma',\rho)) : \rho(opd_L(2)) \neq 0\} \\
\mathit{Trans}_l & \triangleq \{(\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(yes_L,\sigma',\rho)) : \rho(opd_L(1)) \neq 0\} \cup \{(\sigma,m),(L,\sigma',\rho)) \to ((\sigma,m),(no_L,\sigma',\rho)) : \rho(opd_L(2)) \neq 0\} \\
\mathit{Trans}_{r_1} & \triangleq \{((\sigma,m),(L',\sigma'',\rho')(L,\sigma,\rho)) \to ((\sigma,m),(next_L,\sigma,\rho))\} \\
\mathit{Trans}_{r_1} & \triangleq \{((\sigma,m),(L',\sigma'',\rho')(L,\sigma,\rho)) \to ((\sigma,m),(next_L,\sigma,\rho|^v_L)) : v = \rho'(opd_{L'}(1))\} \\
\mathit{Trans}_m & \triangleq \{((\sigma,m),(L,\sigma,\rho)) \to ((\sigma,m),\varepsilon) : proc(L) = \texttt{main}\}
\end{array}
$$

Fig. 3. Transitions Rules defining the semantics of *C-K*

$o : V \to \mathsf{BIT}^k$	the function where $o(v) = 0$ for all $v \in V$
$\ominus(v)$	result of applying the unary operator $\ominus$ to $v \in \mathsf{BIT}^k$
$\oplus(v_1, v_2)$	result of applying the binary operator $\oplus$ to $v_1, v_2 \in \mathsf{BIT}^k$
$f\|_x^v$	function where $f\|_x^v(y) \triangleq \begin{cases} v & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$
$var(l)$	variable at label l
$proc(l)$	procedure where $l \in PROC_{proc(l)}$

Table 4. Notations used in Table 3

(a) Control-Flow		(b) Data-Flow	
$next_0 = 1$	$yes_9 = 10$	$opd_1(1) = \mathtt{n}$	$opd_{11}(1) = 10$
$next_1 = 2$	$no_9 = 12$	$odp_1(2) = 0$	$opd_{14}(1) = 12$
$next_2 = 3$	$next_{10} = 11$	$opd_3(1) = 2$	$opd_{14}(2) = 13$
$next_3 = 4$	$next_{12} = 13$	$opd_4(1) = \mathtt{x}$	$opd_{15}(1) = 14$
$next_4 = 5$	$next_{13} = 14$	$odp_4(2) = 3$	$opd_{17}(1) = 15$
$next_6 = 7$	$next_{14} = 15$	$opd_8(1) = 6$	$opd_{17}(2) = 16$
$next_7 = 8$	$next_{15} = 16$	$opd_8(2) = 7$	$opd_{18}(1) = 17$
$next_8 = 9$	$next_{16} = 17$	$opd_9(1) = 8$	
	$next_{17} = 18$		

Fig. 4. Control– and Data-Flow for the Program in Fig. 2

value. We therefore introduce an *artificial* initial state *start* and the set of start transitions $Trans_s$. The rules in $Trans_c$ show the rules for accessing constants. It changes the value at the register of the current expression to the value of the constant. Similarly, the rules $Trans_v$ for reading values of variables or from the memory (using indirect addresses) change the value of the register associated with the expression to the corresponding value of the global variable (first case), local variable (second case) or memory address (contained in the register of the operand). The rule $Trans_\ominus$ shows the transition rules for the unary operators $\ominus$. According to the ISO *C* language specification, the behavior is undefined, if the result of a mathematical operation is exceptional (such as division by 0) or an overflow occurs. This means that anything can happen (from continuing program execution at any configuration up to exceptional termination). This undefined behavior is modeled by the second set of transition rules of $Trans_\oplus$. The short-circuit operators are associated with two instructions: The first instructions $\mathtt{\&\&}_1$ and $\mathtt{||}_1$ are used for deciding whether the second operand has to be evaluated. The second instructions are used for finally evaluating the second operand. Fig. 5 shows the control- and data-flow of the short-circuit operators.

The execution of assignments stores the value of the right hand side (second operand) to the variable or address defined by the left hand side of the assignment (and thus changes the store for global variables, local variables, or the memory, respectively). The main idea for the transition rules for procedure calls is to push the current procedure frame onto the stack and allocate a new frame for the called procedure. A variable declaration without an initialization expression has no effect. A variable declaration with an initialization expression has the

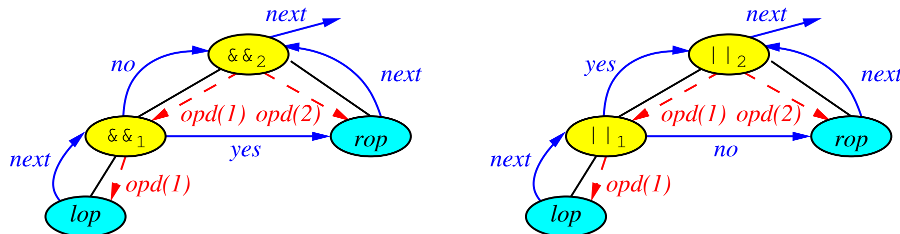


Fig. 5. Control- and Data-Flow for Short-Circuit Operators

same effect as an assignment to this variable (which is always a local variable in this case). The transitions for conditional statements execute the statements of the then-part if the result of the expression is true (beginning at yes_L) and the else-part (if present) or the statement after the conditional (if the else-part is not present), respectively, if the result of the expression is false (in both cases, the execution proceeds at no_L. The transition rule for loops is analogous. The only distinction between conditionals and loops is the definition of yes_L and no_L. The block statement doesn't require a transition rule: If L is the label of the statement preceding the block, then $next_L$ is the first statement within the block. Furthermore, if L is the last statement of the block, $next_L$ is the statement to be executed after the block. The return statement of a proper procedure simply pops the current frame from the stack. The return statement of the function must pass the result of the function. This has been stored at the register for the return expression and must be stored at the register of the function call. Returning from `main` will stop the execution and requires therefore an additional transition rule.

Example 5. Fig. 6 shows a run of the program in Fig. 2. Since we have no pointer variables, we omit the memory.

Remark 2. We have not shown the semantics of allocating objects on the heap m. However, the semantics of `malloc` can easily be implemented by introducing a global variable `void * hp` for storing addresses. It is initialized with the maximal address. With this global variable, it is straightforward to implement `malloc`:

```
void * malloc(int s) {
  hp=hp-s;
  return hp;
}
```

Thus, the semantics of *C-K* is formally described as pushdown system. Since for given configurations κ it is decidable for pushdown systems it is always possible that κ reaches κ' we have the

Corollary 1. *For* C-K*, the halting problem is decidable.*

Remark 3. There is no construct in ISO *C* that prevents modeling of the semantics as a symbolic pushdown system.

start

$$\left(\{n \mapsto 0\}, \left(0, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 20 & -1 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_s$$

$$\left(\{n \mapsto 0\}, \left(1, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & -1 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(2, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & -1 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{=}}$$

$$\left(\{n \mapsto 2\}, \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(6, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 4 & 0 & 2 & 9 & -1 & 22 & 47 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{fak}(n)}$$

$$\left(\{n \mapsto 2\}, \left(7, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 0 & 2 & 9 & -1 & 22 & 47 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(8, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 2 & 9 & -1 & 22 & 47 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(9, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & -1 & 22 & 47 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{<=}}$$

$$\left(\{n \mapsto 2\}, \left(12, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & -1 & 22 & 47 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_i$$

$$\left(\{n \mapsto 2\}, \left(13, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & 2 & 22 & 47 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(14, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & 2 & 1 & 47 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{-}}$$

$$\left(\{n \mapsto 2\}, \left(6, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 2 & 3 & 7 & 0 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{fak(n)}}$$

$$\left(\{n \mapsto 2\}, \left(7, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 3 & 7 & 0 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(8, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 1 & 7 & 0 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(9, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 1 & 1 & 0 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{<=}}$$

$$\left(\{n \mapsto 2\}, \left(10, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 1 & 1 & 0 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 9 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_i$$

$$\left(\{n \mapsto 2\}, \left(11, \{n \mapsto 1\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 1 & 1 & 0 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(15, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 1 & 2 & 1 & 1 & 88 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_c$$

$$\left(\{n \mapsto 2\}, \left(16, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 1 & 2 & 1 & 1 & 1 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_{r_2}$$

$$\left(\{n \mapsto 2\}, \left(17, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 1 & 2 & 1 & 1 & 1 & 2 & 9 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_v$$

$$\left(\{n \mapsto 2\}, \left(18, \{n \mapsto 2\}, \begin{array}{c|c|c|c|c|c|c|c|c|c} 6 & 7 & 8 & 10 & 12 & 13 & 14 & 15 & 16 & 17 \\ \hline 1 & 2 & 0 & 1 & 2 & 1 & 1 & 1 & 2 & 2 \end{array}\right) \left(3, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 5 \end{array}\right)\right) \qquad \mathit{Trans}_{\texttt{*}}$$

$$\left(\{n \mapsto 2\}, \left(4, \{x \mapsto -10\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 2 \end{array}\right)\right) \qquad \mathit{Trans}_{r_2}$$

$$\left(\{n \mapsto 2\}, \left(5, \{x \mapsto 2\}, \begin{array}{c|c|c} 0 & 2 & 3 \\ \hline 2 & 2 & 2 \end{array}\right)\right) \qquad \mathit{Trans}_{d_2}$$

$$(\{n \mapsto 2\}, \varepsilon) \qquad \mathit{Trans}_m$$

Fig. 6. A run of the program in Fig. 2

3.1 Covering the Platform-specific Semantics of *C*

Modeling the semantics of *C-K* as a pushdown system we made several assumptions about the semantics of some language constructs. This is necessary because the language definition of ISO *C* gives only a partial definition of the functional behavior of *C*. We call this language definition the *platform-independent semantics.* The behavior not defined by the language definition can be freely defined by the platform, where a *platform* is defined as all layers used to execute the program, including the compiler up to the target hardware. We call this completion of the language definition the *platform-specific semantics.* The platform-specific semantics of ISO *C* and its challenges for cross-platform testing have been discussed by Wenzel et al. [6].

The reason why ISO *C* does not define the complete functional behavior of the language constructs is simply because of legacy: while the *C* language was originally developed in 1972 by Kernighan and Ritchie at AT&T Bell Labs using an informal (and incomplete) language definition [4, 5] the ANSI committee to develop the *C* language definition was only formed in 1983. Meanwhile a large pool of *C* compilers with different interpretation of several language constructs had arisen. As there was no simple agreement possible of which existing language interpretation is the "right" one, the ANSI *C* language definition had been restricted to a definition of that partial behavior where most *C* compilers agree. In 1990 the ISO adopted the ANSI *C* language definition as the ISO *C* standard and since then took over the further development of the language definition.

The split of the ISO *C* language definition into a platform-independent semantics and a platform-specific semantics has a serious implication for deciding the halting problem of *C* programs: whether a *C* program halts or not may depend on the platform-specific semantics. Thus, even though the halting problem for a *C* program is decidable for any platform-specific semantics, the halting property can become undefined if no specific platform is assumed.

The following list gives just a few examples where the platform-specific semantics has the potential to influence the halting property of some *C* programs:

- Conversion from floating point to integer types: if the value is outside the range of the integer type the result is undefined.
- Conversion of negative values of an integer type into a smaller signed integer type: if the negative value cannot be represented in the smaller type, the behavior is platform-specific.
- Shift operation: whether the shift operations are signed or unsigned is platform-specific.
- Size of base types: the absolute size of the base types is platform-specific.
- Evaluation order of operands: the evaluation order is platform-specific for most operands. One notable exception are short-circuit operations, where the evaluation order is always from left to right.
- Floating-point operations: the behavior in case of exceptional results like division by zero is undefined.

Everything of the ISO *C* language definition with behavior being undefined or implementation-specific belongs to the platform-specific semantics. An interesting property for ISO *C* is that undefined behavior includes the possibility of non-termination. Thus, we also have the

Corollary 2. *If an ISO* C*-program always terminates according to the language definition, i.e., the platform-independent semantics, then it is free of undefined behavior.*

Deciding the halting problem of a *C* program based only on the platform-independent semantics requires to consider all possible instantiations of the platform-specific semantics. Since for any instantiation of the base type ranges a *C* program has a finite state space and, of course, a finite code length, the set of possible instantiations of the platform-specific semantics is also finite for any ranges of the base types. As a consequence we have the

Corollary 3. *The halting problem of an ISO* C *program based only on the platform-independent semantics and upper bounds of the base type ranges is still decidable (with the result being either* yes*,* no*, or* platform-specific*).*

Remark 4. The problem complexity is dramatically higher than in case of considering a concrete platform-specific semantics, since each run of a concrete platform-specific semantics is also a run of the platform-independent semantics but not vice versa.

4 Related Work

Software model checking follows a similar approach [7–12]. Mostly, it abstracts behavior to symbolic descriptions, i.e. variables over a finite range etc. can be defined and they form the state space together with control variables. Popular examples or finite state descriptions are PROMELA for the SPIN model-checker [13] and the SMV-model checker [14, 15]. Remopla is a symbolic description for pushdown systems and used for the Moped model-checker [16, 17]. The semantics of Remopla can be formally defined by mapping it to a pushdown system similar as in this work. A special case of pushdown systems are Boolean programs (i.e. each variable can store only Boolean values) [18]. The decidability of reachability of configurations of pushdown-systems is a well-known result that can be found in textbooks [19]. In [20–23] it is shown that also model-checking on pushdown-systems is possible.

The decidability of program analysis at source-code level has already been the subject of investigation. Previously, it has been shown that *may analysis* and *must analysis* are not decidable using ISO C programs as examples [24, 25]. In contrast, we prove the decidability of ISO *C* programs by considering details of the language definition that make ISO *C* programs inherently finite state and thus decidable.

5 Conclusion

The use of conservative program analysis techniques for approximative analysis results is often justified by citing the undecidability of the halting problem.

In this paper we have shown that a class of real programming languages is decidable. Our approach is based on the assumption that the base types of the language and the heap memory are of finite range, but allowing an unbounded recursion depth of function calls. In case of Java the base types are already bounded, but the potentially infinite heap has to be bounded to a maximum size. In case of ISO *C* we have to bound the ranges of base types. The heap in ISO *C* inherently is of finite size for base type ranges because of the address operator and the finite size of pointer variables. Interestingly, Java Byte code has also a heap of finite size because references must be stored in the operand stack and the entries have the lengths of machine words.

We have proven the decidability of such programming languages by showing how to transform them into a pushdown automaton, for which the decidability is a well-known result. As a concrete example we have selected a representative subset of ISO *C*.

Finally, we have got the interesting insight that the halting property for ISO *C* can depend on the concrete system platform, i.e., implementation details not specified by the language definition can influence the halting of a program.

References

1. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckman, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The worst-case execution time problem - overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) **7** (2008)
2. Kirner, R., Puschner, P.: Obstacles in worst-cases execution time analysis. In: Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing, Orlando, Florida (2008) 333–339
3. Organisation, I.S.: ISO/IEC 9899:1999 Programming Languages - C. 2nd edn. American National Standards Institute, New York (1999) Technical Committe: JTC 1/SC 22/WG 14.
4. Kernighan, B.W., Ritchie, D.M.: The C programming language. Prentice Hall Press, Upper Saddle River, NJ (1989)
5. Ritchie, D.M.: The Development of the C Programming Language. In: History of programming languages—II. ACM Press, New York, NY, USA (1996) 671–698
6. Wenzel, I., Rieder, B., Kirner, R., Puschner, P.: Cross-platform verification framework for embedded systems. In: Proc. 5th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'07), Santorini Island, Greece (2007)
7. Obdrzalek, J. In: Model Checking Java Using Pushdown Systems. LFCS, University of Edinburgh (2002)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G. In: Software Verification with BLAST. In 10th International Workshop on Model Checking of Software (SPIN), LNCS Volume 2648, 235-239. Springer (2003)

9. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F. In: Model Checking Programs. Automated Software Engineering Volume 10(2), 203-232, Kluwer Academic (2003)
10. Ivancic, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, C., Yang, Z. In: Model Checking C Programs Using F-SOFT. Proc. of the International Conference on Computer Design (ICCD) (2005)
11. Suwimonteerabuth, D., Schwoon, S., Esparza, J. In: jMoped: A Java Bytecode Checker Based on Moped. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, Springer (2005)
12. Berger, F. In: A test and verification environment for Java programs. Diplomarbeit Nr. 2470, Universitt Stuttgart (2006)
13. G. J. Holzmann: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
14. McMillan, K.: Symbolic Moldel Checking. Kluwer Academic Publishers (1993)
15. Jin Yang, Andreas Tiemeyer: Lazy symbolic model checking. In: DAC 00: Proceedings of the 37th conference on Design automation, ACM Press New York (2000) 35–38
16. Esparza, J., Schwoon, S. In: A BDD-based model checker for recursive programs. LNCS Volume 2102, 324-336, Springer (2001)
17. Schwoon, S. In: Model-Checking Pushdown Systems. Technische Universitt Mnchen (2002)
18. Ball, T., Rajamani, S.K. In: Bebop: A Symbolic Model Checker for Boolean Programs. 7th International SPIN Workshop, LNCS Volume 1885, 113-130, Springer (2000)
19. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. 2nd edn. Addison Wesley (2001)
20. Bouajjani, A., Esparza, J., Maler, O. In: Reachability Analysis of Pushdown Automata: Application to Model-Checking. Proc. of the 8th International Conference on Concurrency Theory, LNCS 1243 (1997)
21. Walukiewicz, I. In: Model checking CTL Properties of Pushdown Systems. In FSTTCS'00, LNCS 1974 (2000)
22. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S. In: Efficient algorithms for model checking pushdown systems. Proc. of the 12th International Conference on Computer Aided Verification, LNCS 1855 (2000)
23. Esparza, J., Kucera, A., Schwoon, S. In: Model-Checking LTL with Regular Valuations for Pushdown Systems. Proc. of the 4th International Symposium on Theoretical Aspects of Computer Software, LNCS 2215 (2002)
24. Landi, W.: Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS) **1** (1992) 323–337
25. Ramalingam, G.: The undecidability of aliasing. ACM Transactions on Programming Languages and Systems (TOPLAS) **16** (1994) 1467–1471

From Trusted Annotations to Verified Knowledge*

Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec, and Markus Schordan

Vienna University of Technology, Institute of Computer Languages, Austria,
{adrian,knoop}@complang.tuwien.ac.at
Vienna University of Technology, Institute of Computer Engineering, Austria,
{raimund,albrecht}@vmars.tuwien.ac.at
University of Applied Sciences Technikum Wien, Austria,
schordan@technikum-wien.at

Abstract. WCET analyzers commonly rely on user-provided annotations such as loop bounds, recursion depths, region- and program constants. This reliance on user-provided annotations has an important drawback. It introduces a Trusted Annotation Basis into WCET analysis without any guarantee that the user-provided annotations are safe, let alone sharp. Hence, safety and accuracy of a WCET analysis cannot be formally established. In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding sharper (tighter) time bounds. Fundamental to our approach is to apply model checking in concert with other more inexpensive program analysis techniques, and the coordinated application of two algorithms for *Binary Tightening* and *Binary Widening*, which control the application of the model checker and hence the computational costs of the approach. Though in this paper we focus on the control of model checking by Binary Tightening and Widening, this is embedded into a more general approach in which we apply an array of analysis methods of increasing power and computational complexity for proving or disproving relevant time bounds of a program. First practical experiences using the sample programs of the Mälardalen benchmark suite demonstrate the usefulness of the overall approach. In fact, for most of these benchmarks we were able to empty the trusted annotation base completely, and to tighten the computed WCET considerably.

* This paper has been published in the Proceedings of the 9th Int'l Workshop on Worst-Case Execution Time Analysis (WCET'09). This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (CoSTA) under contract No P18925-N13 and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

1 Motivation

The computation of loop bounds, recursion depths, region- and program constants is undecidable. It is thus commonly accepted that WCET analyzers rely to some extent on user-assistance for providing bounds and constants. Obviously, this is tedious, complex, and error-prone. State-of-the-art approaches to WCET analysis thus provide for a fully automatic preprocess for computing required bounds and constants using static program analysis. This unburdens the user since his assistance is reduced to bounds and constants, which cannot automatically be computed by the methods employed by the preprocess. Typically, these are classical data-flow analyses for constant propagation and folding, range analysis and the like, which are particularly cost-efficient but may fail to verify a bound or the constancy of a variable or term. WCET analyzers then rely on user-assistance to provide the missing bounds which are required for completing the WCET analysis. This introduces a *Trusted Annotation Base* (*TAB*) into the process of WCET analysis. The correctness (safety) and optimality (tightness) of the WCET analysis depends then on the safety and tightness of the bounds of the TAB provided by the user.

In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding sharper (tighter) time bounds.

Figure 1 illustrates the general principle of our approach. At the beginning the entire annotation base that is added by the user where static analysis fails to establish the required information, is assumed and trusted to be correct, thus we call it *Trusted Annotation Base* (*TAB*). Using model checking we aim to verify as many of these user-provided facts as possible. In this process we shrink the trusted fraction of the annotation base and establish a growing verified annotation base. In Figure 1 the current state in this process is visualized as the horizontal bar. In our approach we are lowering this bar, representing the decreasing fraction of trust to an increasing fraction of verified knowledge, and thus transfer *trusted user-belief* into *verified knowledge*.

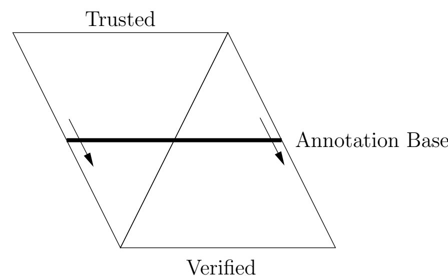


Fig. 1. The annotation base: shrinking the trusted annotation base and establishing verified knowledge about the program

2 Shrinking the Trusted Annotation Base – Sharpening the Time Bounds

2.1 Shrinking the Trusted Annotation Base

The automatic computation of bounds by the preprocesses of up-to-date approaches to WCET analysis is a step towards keeping the trusted annotation base small. In our approach we go a step further to shrinking the trusted annotation base. In practice, we often succeed to empty it completely.

A key observation is that a user-provided bound – which the preprocessing analyses were unable to compute – can not be checked by them either. Hence, verifying the correctness of the corresponding user annotation in order to move it *a posteriori* from the trusted annotation base to the verified knowledge base requires another, more powerful and usually computationally more costly approach. For example, there are many algorithms for the detection of copy constants, linear constants, simple constants, conditional constants, up to finite constants detecting different classes of constants at different costs [7]. This provides evidence for the variety of available choices for analyses using the example of constant propagation and folding. While some of these algorithms might in fact well be able to verify a user annotation, none of these algorithms is especially prepared and suited for solely verifying a data-flow fact at a particularly chosen program location, a so-called *data-flow query*. This is because these algorithms are exhaustive in nature. They are designed to analyze whole programs. They are not focused towards deciding a data-flow query, which is the domain of *demand-driven program analyses* [3, 6]. Like for the more expressive variants of constant propagation and folding, however, demand-driven variants of program analyses are often not available.

In our approach, we thus propose to use *model checking* for the *a posteriori* verification of user-provided annotations. Model checking is tailored for the verification of data-flow queries. Moreover, the development of software model checkers made tremendous progress in the past few years and are now available off-the-shelf. The Blast [1] and the CBMC [2] model checkers are two prominent examples. In our experiments reported in Section 4 we used the CBMC model checker.

The following example demonstrates the ease and elegance of using a model checker to verify a loop bound, which we assume could not be automatically bounded by the program analyses used. The program fragment on the left-hand side of Figure 2 shows a loop together with a user-provided annotation of the loop. The program on the right-hand side shows the transformed program which is presented to CBMC to verify or refute the user-provided annotation shown in the program on the right-hand side:

In this example the CBMC model checker comes up with the answer "yes," i.e., the loop bound provided by the user is safe; allowing thus for its movement from the trusted to the verified annotation base. If, however, the user were to provide $__bound \leq 3$ as annotation, model checking would fail and produce a

```
int binary_search(int x) {
  int fvalue, mid, low = 0, up = 14;
  fvalue = (-1); /* all data are positive */


  while(low <= up){
#pragma wcet_trusted_loopbound(7)
    mid = low + up >> 1;
    if (data[mid].key == x) { /* found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else if (data[mid].key > x)
    up = mid - 1;
    else low = mid + 1;


  }


  return fvalue;
}
```

→

```
int binary_search(int x) {
  int fvalue, mid, low = 0, up = 14;
  fvalue = (-1); /* all data are positive */

  unsigned int __bound = 0;
  while(low <= up){

    mid = low + up >> 1;
    if (data[mid].key == x) { /* found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else if (data[mid].key > x)
    up = mid - 1;
    else low = mid + 1;

    __bound += 1;
  }
  assert(__bound <= 7);

  return fvalue;
}
```

Fig. 2. Providing loop bound annotations for the model checker

counter example as output. Though negative, this result would still be most valuable. It allows for preventing usage of an unsafe trusted annotation base in a subsequent WCET analysis. Note that the counter example itself, which in many applications is the indispensable and desired output of a failed run of a model checker, is not essential for our application. It might be useful, however, to present it to the user when asking for another candidate of a bound, which can then be subject to *a posteriori* verification in the same fashion until a safe bound is eventually found.

Next we introduce a more effective approach to come up with a safe and even tight bound, if so existing, which does not even rely on any user interaction. Fundamental for this are the two algorithms *Binary Tightening* and *Binary Widening* and their coordinated interaction. The point of this coordination is to make sure that model checking is applied with care as it is computationally expensive.

2.2 Sharpening the Time Bounds

Binary Tightening Suppose a loop bound has been proven safe, e.g. by verifying a user-provided bound by model checking or by a program analysis. Typically, this bound will not be tight. In particular, this will hold for user-provided bounds. In order to exclude channeling an unsafe bound into the trusted annotation base, the user will generously err on the side of caution when providing a bound. This suggests the following iterative approach to tighten the bound, which is an application of the classical pattern of a binary search algorithm, thus called *binary tightening* in our scenario.

Let b_0 denote the value of the initial bound, which is assumed to be safe. Per definition b_0 is a positive integer. Then: call procedure *binaryTightening* with

the interval $[0..b_0]$ as argument, where $binaryTightening([low..high])$ is defined as follows:

1. Let $m = \lceil \frac{low+high}{2} \rceil$.
2. ModelCheck(m is a safe bound):
3. yes: $low = m$: **return** m
   $low = m - 1$: ModelCheck(low is a safe bound)
   yes: **return** low no: **return** m
   otherwise: $binaryTightening([low..m])$
4. no: $high = m$: **return** $false$
   $high = m + 1$: ModelCheck($high$ is a safe bound)
   yes: **return** $high$ no: **return** $false$
   otherwise: $binaryTightening([m..high])$

Obviously, $binaryTightening$ terminates. If it returns $false$, a safe bound tighter than that of the initial bound b_0 could not be established. Otherwise, i.e., if it returns value b, this value is the least safe bound. This means b is tight. If it is smaller than b_0, we succeeded to sharpen the bound.

Binary widening described next allows for proceeding in the case where a safe bound is not known *a priori*. If a safe bound (of reasonable size) exists, binary widening will find one, without any further user interaction.

Binary Widening Binary widening is dual to binary tightening. Its functioning is inspired by the risk-aware gambler playing roulette, who exclusively bets on 50% chances like red and black. Following this strategy, in principle, any loss can be flattened by doubling the bet the next game. In reality, the maximum bet allowed by the casino or the limited monetary resources of the gambler, whatever is lower, prevent this strategy to work out in reality. Nonetheless, the idea of an externally given limit yields the inspiration for the *Binary Widening* algorithm to avoid looping if no safe bound exists. A simple approach is to limit the number of recursive calls of binary widening to a predefined maximum number. The version of binary widening we present below uses a different approach. It comes up with a safe bound, if one exists, and terminates, if the size of the bound is too big to be reasonable, or does not exist at all. This directly corresponds to the limit set by a casino to a maximum bet.

Let b_0 be an arbitrary number, $b_0 \geq 1$, and let max be the maximum value for a safe bound considered reasonable. Then: Call procedure $binaryWidening$ with b_0 and max as arguments, where $binaryWidening(b, limit)$ is defined as follows:

1. if $b > limit$: **return** false
2. ModelCheck(b is a safe bound):
3. yes: **return** b
4. no: $binaryWidening(2 * b, limit)$

Obviously, $binaryWidening$ terminates.[1] If it returns $false$, at most a bound of a size considered unreasonably big exists, if at all. Otherwise, i.e., if it returns value b, this value is a safe bound. The ratio behind this approach is the following: if a safe bound exists, but exceeds a predefined threshold, it can be considered practically useless. In fact, this scenario might indicate a programming error and should thus be reported to the programmer for inspection. A more refined approach might set this threshold more sophisticatedly, by using application dependent information, e.g., such as a coarse estimate of the execution time of a single execution of the loop and a limit on the overall execution time this loop shall be allowed for.

Coordinating Binary Tightening and Widening Once a safe bound has been determined using binary widening, binary tightening can be used to compute the uniquely determined safe tight bound. Because of the exponential resp. logarithmic behaviour in the selection of arguments for binary widening and tightening, model checking is called moderately often. This is the key for the practicality of our approach, which we implemented in our WCET analyzer TuBound, as described in Section 3. The results of practical experiments we conducted with the prototype implementation are promising. They are reported in Section 4.

3 Implementation within TuBound

3.1 TuBound

TuBound [9] is a research WCET analyzer tool working on a subset of the C++ language. It is unique for uniformly combining static program analysis, optimizing compilation and WCET calculation. Static analysis and program optimization are performed on the abstract syntax tree of the input program. TuBound is built upon the SATIrE program analysis framework [12] and the TERMITE program transformation environment.[2] TuBound features an array of algorithms for loop analysis of different accuracy and computation cost including sophisticated analysis methods for nested loops. A detailed account of these methods can be found in [8].

3.2 Implementation

The Binary Widening/Tightening algorithms are implemented by means of a dedicated TERMITE source-to-source transformer T. For simplicity and uniformity we assume that all loops are structured. In our implementation unstructured goto-loops are thus transformed by another source-to-source transformer

[1] In practice, the model checker might run out of memory before verifying a bound, if it is too large, or may take too much time for completing the check.

[2] `http://www.complang.tuwien.ac.at/adrian/termite`

T' into while-loops beforehand, where possible. On while-loops the transformer T works by locating the first occurrence of a `wcet_trusted_loopbound(N)` annotation in the program source and then proceeding to rewrite the encompassing loop as illustrated in the example of Figure 3.[3] Surrounding the loop statement,

```
assertions(..., Statement, AssertedStatement) :-
  Statement = while_stmt(Test, basic_block(Stmts, ...), ...),
  get_annot(Stmts, wcet_trusted_loopbound(N), _),

  counter_decl('__bound', ..., CounterDecl),
  counter_inc('__bound', ..., Count),
  counter_assert('__bound', N, ..., CounterAssert),

  AssertedStatement =
    basic_block([CounterDecl,
                 while_stmt(Test, basic_block([Count|Stmts], ...), ...),
                 CounterAssert], ...).
```

Fig. 3. Excerpt from the source-to-source transformer T

a new compound statement is generated, which accommodates the declaration of a new unsigned counter variable which is initialized to zero upon entering the loop. Inside the loop, an increment statement of the counter is inserted at the very first location. After the loop, an assertion is generated which states that the count is at most of value N, where this value is taken from the annotation (cf. Figure 2).

The application of the transformer is controlled by a driver, which calls the transformer for every trusted annotation contained in the source code. Depending on the result of the model checker and the coordinated application of the algorithms for binary widening and tightening, the value and the status of each annotation is updated. In the positive case, this means the status is changed from *trusted annotation* to *verified knowledge*, and the value of the originally trusted bound is replaced by a sharper, now verified bound. Figure 4 shows a snapshot of processing the *janne_complex* benchmark. In this figure, the status and value changes are highlighted by different colors.

4 Experimental Results

We implemented our approach as an extension of the TuBound WCET analyzer and applied the extended version to the well-known Mälardalen WCET benchmark suite. As a baseline for comparison we used the 2008 version of TuBound, which took part in the 2008 WCET Tool Challenge [5], later on called the basic version of TuBound. In the spirit of the WCET Tool Challenge [4, 5] we do encourage authors of other WCET analyzers to carry out similar experiments.

[3] For better readability, the extra arguments containing file location and other bookkeeping information are replaced by "...".

```
...
int complex(int a, int b)
{
  while(a < 30) {
#pragma wcet_trusted_loopbound(30)
      while(b < a) {
#pragma wcet_trusted_loopbound(30)
        if (b > 5)
          b = b * 3;
        else
          b = b + 2;
        if (b >= 10 && b <= 12)
          a = a + 10;
        else
          a = a + 1;
      }
      a = a + 2;
      b = b - 10;
    }
  return 1;
}
...
```

Containing two trusted loop annotations

→

```
...
int complex(int a, int b)
{
    while(a < 30) {
#pragma wcet_loopbound(16)
      {
        unsigned int __bound = 0U;
        while(b < a){
#pragma wcet_trusted_loopbound(30)
          ++__bound;

         if (b > 5)
           b = b * 3;
         else
           b = b + 2;
         if (b >= 10 && b <= 12)
           a = a + 10;
         else
           a = a + 1;
      }
      assert(__bound <= 30U);
    }
    a = a + 2;
    b = b - 10;
  }
  return 1;
}
...
```

Outer loop annotation verified and tightened, inner being checked

Fig. 4. Illustrating trusted bound verification and tightening

Our experiments conducted were guided by two questions: “Can the number of automatically bounded loops be significantly increased?” and “How expensive is the process?”. The benchmarks were performed on a 3 GHz Intel Xeon processor running 64-bit Linux. The model checker used was CBMC 2.9, which we applied to testing loop bounds up to the size of $2^{13} = 8192$ using a timeout of 120 seconds and a maximum unroll factor of $2^{13} + 1$. The “compress” and “whet” benchmarks contained unstructured goto-loops; as indicated in Section 3.2 these were automatically converted into do-while loops beforehand by a source-to-source transformation.

Our findings are summarized in Table 1. Column three of this table shows the percentage of loops that can be bounded by the basic version of TuBound; column four shows the total percentage of loops the extended version of TuBound was able to bound. The last column shows the accumulated runtime of the model checker for the remaining loops.

Comparing columns three and four reveals the superiority of the extended version of TuBound over its basic variant. The extended version succeeds to bound 67% of the loops the basic version could not bound.

Considering column five, it can be seen that the model checker terminates quickly on small problems but that the runtime and space requirements can increase to practically infeasible amounts on problems suffering from the state explosion problem. Such a behaviour can be triggered, if the initialization values which are part of the majority of the Mälardalen benchmarks are manually

Benchmark	Loops	TuBound basic	with Model Checking	Runtime
bs	1	0.0%	100.0%	0.03s
duff	2	50.0%	50.0%	0s
fft1	11	54.5%	81.8%	0.43s
janne_complex	2	0.0%	100.0%	0.18s
minver	17	94.1%	100.0%	0.06s
nsichneu	1	0.0%	100.0%	5.59s
qsort-exam	6	0.0%	66.6%	0.02s
statemate	1	0.0%	100.0%	0.06s
whet	11	90.9%	90.9%	0s
adpcm	18	83.3%	83.3%	timeout
compress	8	25.0%	25.0%	timeout
fir	2	50.0%	50.0%	timeout
insertsort	2	0.0%	0.0%	timeout
lms	10	60.0%	60.0%	timeout
select	4	0.0%	0.0%	timeout
bsort100	3	100.0%	100.0%	–
cnt	4	100.0%	100.0%	–
cover	3	100.0%	100.0%	–
crc	3	100.0%	100.0%	–
edn	12	100.0%	100.0%	–
expint	3	100.0%	100.0%	–
fdct	2	100.0%	100.0%	–
fibcall	1	100.0%	100.0%	–
jfdctint	3	100.0%	100.0%	–
lcdnum	1	100.0%	100.0%	–
ludcmp	11	100.0%	100.0%	–
matmult	5	100.0%	100.0%	–
ndes	12	100.0%	100.0%	–
ns	4	100.0%	100.0%	–
qurt	1	100.0%	100.0%	–
sqrt	1	100.0%	100.0%	–
st	5	100.0%	100.0%	–
recursion	0	–	–	–
Total Percentage		77.0%	84.7%	

Table 1. Results for the Mälardalen benchmarks

invalidated by introducing a faux dependency on e.g. `argc`. This demonstrates that model checking is to be used with care or the model checker be fed with additional information guiding and simplifying the verification task.

The fully-fledged variant of our approach, which we highlight in the next section is tailored towards this goal.

5 Extensions: The Fully Fledged Approach

The shrinking of the trusted annotation base and sharpening of time bounds, as described in Section 2, is based on model checking. Based on our experience, we believe that the model checking approach can be especially valuable in the real world when (i) it is combined with advanced program slicing techniques to reduce the state space and (ii) the results of static analyses (like TuBound's variable-interval analysis) are used to narrow the value ranges of variables, thus regaining a feasible problem size. This leads to the following extension of our approach to improve efficiency:

1. By using a pool of analysis techniques with different computational complexity: As shown in Figure 5, model checking is considered as one of the most complex analysis methods. On the other side, techniques like *constant propagation* or *interval analysis* are relatively fast. Thus we are interested in exploiting the fast techniques wherever beneficial and using the relatively complex techniques rarely.
2. By using a smart activation mechanism for the different analysis techniques: As shown on the right of Figure 5 we are interested in the interaction of the different analysis techniques. We do not aim to use the pool of analysis techniques in waves of different complexity, i.e., first applying the fast techniques and then gradually shifting towards the more complex techniques. Instead we aim for a smart interaction of the different analysis techniques. For example, whenever a technique with relatively high computational complexity has been applied, we can again apply techniques of relatively simple complexity to compute the closure of flow information based on previously obtained results; thus *squeezing* the annotation base.

We also think that profiling techniques are useful to guide the heuristics to be used within our static analysis techniques. For example, execution samples obtained by profiling can be used to elicit propositions to be verified by model checking.

The fully fledged approach envisioned in this section provides the promising potential as a research platform for complementing program analysis techniques.

6 Conclusions

Model checking has been used before in the context of WCET analyzers. Examples of our own related work are the ForTAS [13], MoDECS [14], and ATDGEN

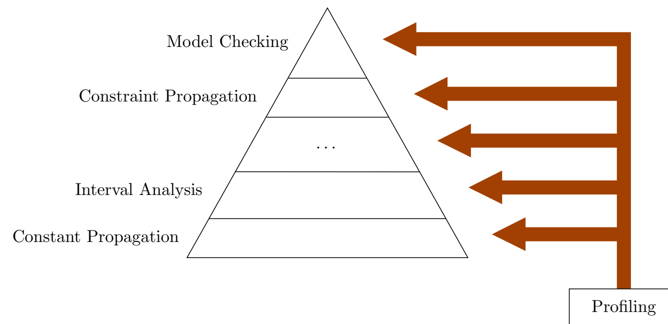


Fig. 5. Pool of complementary analysis techniques with different complexity

projects [10, 11], which are concerned with measurement-based WCET analysis. In these three projects, model checking is used to generate test data for the execution of specific program paths. Intuitively, in these applications the model checker is presented with formulae stating that a specific program path is infeasible. If these formulae can be refuted by the model checker, the counter examples generated provide the test data ensuring the execution of the particular paths. Otherwise, the paths are known to be infeasible. Hence, the search for test data is in vain. In these applications the counter examples generated in the course of failing model checker runs are the truly desired output, whereas successful runs are of less interest stopping just the search for test data for the path under consideration. This is in contrast and opposite to our application of shrinking the trusted annotation base. In our application, the counter example of a failed model checker run is of little interest. We are interested in successful runs of the model checker as they allow us to change a *trusted annotation* into *verified knowledge*. This opens a new application domain for model checking in the field of WCET analysis. Our preliminary practical results demonstrate the practicality and power of this approach.

References

1. Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, October 2007.
2. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
3. Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992 – 1030, 1997.
4. Jan Gustafsson. The WCET tool challenge 2006. In *Preliminary Proc. 2nd International IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.

5. Niklas Holsti, Jan Gustafsson, Guillem Bernat (eds.), Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET Tool Challenge 2008: Report. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 149–171, Prague, Czech Republic, July 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3.
6. Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE-3*), pages 104 – 115, 1995.
7. Steven S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4.
8. Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level wcet analysis. In *Proc. 18th International Workshop on Logic-based methods in Programming Environments (WLPE 2008)*, pages 77–89, Udine, Italy, December 2008.
9. Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound – a conceptually new tool for worst-case execution time analysis. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 141–148, Prague, Czech Republic, July 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3.
10. Bernhard Rieder. *Measurement-Based Timing Analysis of Applications written in ANSI-C.* PhD thesis, Technische Universität Wien, Vienna, Austria, 2009 (forthcoming).
11. Bernhard Rieder, Ingomar Wenzel, and Peter Puschner. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement-based WCET analysis. In *Proc. 6th International Workshop on Intelligent Solutions in Embedded Systems (WISES 2008)*, Regensburg, Germany, July 2008.
12. Markus Schordan. Source-To-Source Analysis with SATIrE – an Example Revisited. In *Proceedings of Dagstuhl Seminar 08161: Scalable Program Analysis*. Germany, Dagstuhl, April 2008.
13. Vienna University of Technology and TU Darmstadt. The ForTAS project. Web page (`http://www.fortastic.net`). Accessed in June 2009.
14. Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. In *Proc. 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, October 2008.

Tree Automata for Analyzing Dynamic Pushdown Networks

Peter Lammich

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
peter.lammich@uni-muenster.de

Abstract. Dynamic Pushdown Networks (DPNs) are an abstract model for concurrent programs with recursive procedures and dynamic process creation. Usually, DPNs are described with an interleaving semantics, where an execution is a sequence of steps. Recently, we introduced a true-concurrency semantics for DPNs, where executions are trees.
The standard analysis methods for DPNs are based on a saturation algorithm, that, given a set of configurations, computes the set of all predecessor configurations. In this paper we present an alternative analysis algorithm that is based on tree automata. DPN executions as well as the properties to be analyzed are represented as tree automata, and the analysis is done by standard tree-automata algorithms for intersection and emptiness check.

1 Introduction

Many existing analysis techniques for concurrent programs use parallel pushdown automata (e.g. [5, 3, 4]) or parallel procedure calls (e.g. [10, 8]) as a model for concurrency. However, there are programming languages, like Java, that support dynamic thread creation. This cannot be mapped to parallel procedure calls [2], and it can only be mapped to parallel pushdown systems when bounding the maximum number of allowed threads. In contrast, dynamic pushdown networks (DPNs)[2] natively support dynamic thread creation.

DPNs extend pushdown systems by rules with the side-effect of creating a new pushdown process that is then executed concurrently. In [2], analysis of DPNs is done by automata based techniques. The key result for the analysis of DPNs is, that pre^*_M-computation preserves regularity. This can be used for, e.g., bitvector kill/gen analysis [2] or bounded model checking of DPNs with shared memory [1].

Originally, DPNs are given an interleaving semantics, where an execution is a totally ordered sequence of steps. More recently, a true concurrency semantics for DPNs has been introduced [7], where executions are modeled as trees. It is shown how to decide tree-regular properties of such executions, and thus generalize the results on DPNs of [2]. Moreover, this technique can be used to precisely compute pre^*_M-sets for DPNs with well-nested locks [7].

The techniques presented in [7] compute the cross-product of a DPN and a tree automaton for the regular property to be analyzed. This cross-product is, again, a DPN and can be analyzed using the standard pre^*_M-computation [2]. In this paper, we explore a different approach for the analysis of DPNs. We also use the true-concurrency semantics presented in [7], but instead of computing the cross-product DPN, we use tree-automata techniques: Let M be a DPN with the set of executions e_M, and Φ be a regular property. We want to check whether the DPN has an execution with property Φ, i.e. whether $e_M \cap \Phi \neq \emptyset$. Our analysis computes a tree-automaton A with language $L(A)$ from the DPN, such that $e_M = \alpha(L(A))$, where α maps a tree of $L(A)$, called *regular execution tree*, to an execution of the true-concurrency semantics, and $\alpha(L(A))$ is the element-wise application of α to the set $L(A)$. The problem now reduces to checking whether $L(A) \cap \alpha^{-1}(\Phi) \neq \emptyset$. If $\alpha^{-1}(\Phi)$ is also a regular set, this can be decided using standard tree automata algorithms.

In this paper, we manually derive the tree automata representation of $\alpha^{-1}(\Phi)$ for a specific reachability property Φ. It is left future work to automatically derive the tree automata representations of $\alpha^{-1}(\Phi)$ from a suitable representation of Φ, e.g. using theory related to macro tree-transducers.

The rest of this paper is organized as follows: In Section 2, we introduce DPNs along with their interleaving and true-concurrency semantics. In Section 3, we introduce regular execution trees, justify them w.r.t. the true-concurrency semantics and derive a specific reachability property for regular execution trees. Finally, in Section 4, we give a short conclusion and outlook to future work.

2 Dynamic Pushdown Networks

A DPN [2] is a tuple $M = (P, \Gamma, L, \Delta)$, where P is a finite set of control states, Γ is a finite stack alphabet with $P \cap \Gamma = \emptyset$, L is a finite set of rule labels, and $\Delta = \Delta_N \cup \Delta_S$ is a finite set of *non-spawning* (Δ_N) and *spawning* rules (Δ_S). A non-spawing rule $p\gamma \overset{l}{\hookrightarrow} p'w \in \Delta_N \subseteq P\Gamma \times L \times P\Gamma^*$ enables a transition on a single pushdown process with state p and top of stack γ to new state p' and new top of stack $w \in \Gamma^*$. A spawning rule $p\gamma \overset{l}{\hookrightarrow} p'w \rhd p_s w_s \in \Delta_S \subseteq P\Gamma \times L \times P\Gamma^* \times P\Gamma^*$ is a pushdown rule with the additional side-effect of creating a new process with initial state p_s and initial stack w_s. DPNs are able to model programs with potentially recursive procedures: A rule with $|w| = 0$ models a return from a procedure, a rule with $|w| = 1$ models a step inside a procedure, and a rule with $|w| = 2$ models a procedure call. Rules with $|w| > 2$ can be seen as generalized procedure calls where more than one return address is pushed onto the stack. In this paper, we restrict DPN rules to be of one of the following forms:

Assumption 1 *Let $r \in \Delta$ be a DPN-rule. Then there exists $p, p', p_s \in P$, $\gamma, \gamma_1, \gamma_2, \gamma', \gamma_s \in \Gamma$ and $l \in L$ such that r has one of the following forms:*

basic step $r = p\gamma \overset{l}{\hookrightarrow} p'\gamma'$

procedure call $r = p\gamma \overset{l}{\hookrightarrow} p'\gamma_1\gamma_2$
procedure return $r = p\gamma \overset{l}{\hookrightarrow} p'$
process creation $r = p\gamma \overset{l}{\hookrightarrow} p'\gamma' \rhd p_s\gamma_s$

Note that these rule types are sufficient to model interprocedural programs with DPNs. Moreover, by replacing generalized calls by sequences of procedure calls (adding new states and stack symbols as necessary), one can transform any DPN into a DPN that satisfies Assumption 1, while preserving the relevant properties for program analysis, e.g. reachability properties. For the rest of this paper, we assume that we have a fixed DPN $M = (P, \Gamma, L, \Delta)$, and a fixed initial configuration $p_0\gamma_0 \in P\Gamma$. Note that this restricts our analysis to programs with one initial process. If programs with more than one initial process are required, one has to create a process that sets up the program's initial configuration.

Interleaving Semantics. We briefly recall the interleaving semantics of a DPN as presented in [2]: Configurations $\mathsf{Conf} := (P\Gamma^*)^*$ are sequences of words from $P\Gamma^*$, each word containing the control state and stack of one of the processes running in parallel. The step relation $\rightarrow \subseteq \mathsf{Conf} \times L \times \mathsf{Conf}$ is the least solution of the following constraints:

$$
\begin{array}{lll}
[\text{nospawn}] & c_1(p\gamma r)c_2 \xrightarrow{l} c_1(p'wr)c_2 & \text{if } p\gamma \overset{l}{\hookrightarrow} p'w \in \Delta_N \\
[\text{spawn}] & c_1(p\gamma r)c_2 \xrightarrow{l} c_1(p_s w_s)(p'wr)c_2 & \text{if } p\gamma \overset{l}{\hookrightarrow} p'w \rhd p_s w_s \in \Delta_S
\end{array}
$$

A [nospawn]-step corresponds precisely to a pushdown operation (manipulating the control state and the top of the stack), a [spawn]-step additionally creates a new process that is inserted to the left of the creating process. We define $\rightarrow^* \subseteq \mathsf{Conf} \times L^* \times \mathsf{Conf}$ to be the reflexive, transitive closure of $\rightarrow$. This semantics is an *interleaving semantics*, because steps of processes running in parallel are interleaved. It models the possible executions on a single processor, where preemption may occur after any step.

In order to simplify the presentation of the analysis, we assume that no configuration with an empty stack is reachable from the initial process $p_0\gamma_0$:

Assumption 2 $\forall l \in L,\ c' \in \mathsf{Conf}.\ p_0\gamma_0 \xrightarrow{l}{}^{*} c' \implies c' \in (P\Gamma^+)^*$

Again, any DPN can be transformed to satisfy this assumption while preserving the relevant properties. The transformation adds a new stack symbol $\gamma_\perp$ at the bottom of each stack.

Also note that a DPN contains no rules to remove an existing process from the configuration. Thus we have:

Lemma 3. $\forall pw \in P\Gamma^*,\ l \in L,\ c' \in \mathsf{Conf}.\ p\gamma \xrightarrow{l}{}^{*} c' \implies c' \in (P\Gamma^*)^+$

In DPNs, termination of a process can be modeled, e.g., by going into a special state from that there is no more progress.

Predecessor Computation. Given a set $C' \subseteq \mathsf{Conf}$ of configurations, the set $\mathsf{pre}_M(C') := \{c \mid \exists c' \in C',\ l \in L.\ c \xrightarrow{l} c'\}$ is the set of *immediate predecessors* of C', i.e. the set of configurations that can make a transition to a $c' \in C'$ in exactly one step. Similarly, $\mathsf{pre}^*_M(C') := \{c \mid \exists c' \in C',\ \bar{l} \in L^*.\ c \xrightarrow{\bar{l}}{}^* c'\}$ is the set of *predecessors* of C', i.e. the set of configurations that can make a transition to a $c' \in C'$ by executing an arbitrary number of steps.

An important result on DPNs is that pre_M and pre^*_M preserve regularity, i.e. if C' is a regular set then $\mathsf{pre}_M(C')$ and $\mathsf{pre}^*_M(C')$ are regular as well, and given an automaton accepting C', automata accepting $\mathsf{pre}_M(C')$ and $\mathsf{pre}^*_M(C')$, respectively, can be computed in polynomial time [2]. This result is the key to analysis of DPNs. For example, let $p\gamma \in P\Gamma$ be a state where a resource is write-accessed. Let $p'\gamma' \in P\Gamma$ be another state where the same resource is read-accessed. The regular set

$$C := (S^* p\gamma \Gamma^* S^* p'\gamma' \Gamma^* S^*) \mid (S^* p'\gamma' \Gamma^* S^* p\gamma \Gamma^* S^*)$$

with $S = P\Gamma^*$ contains exactly those configurations where one process is at state $p\gamma$ and another process is at state $p'\gamma'$. Thus, the query $p_0\gamma_0 \in \mathsf{pre}^*_M(C)$ decides whether a conflict situation between the two resource accesses at $p\gamma$ and $p'\gamma'$ is reachable[1].

Tree Semantics. We recently presented a tree-based semantics for DPNs [7]. An execution of the interleaving semantics is a sequence of steps, inducing a total ordering on the steps. Even steps of independent processes are ordered. In the tree-based semantics, an execution is a tree of steps, only inducing an ordering on steps of the same process and on steps of a created process to come after the step that created the process.

Formally, we model an execution starting at a single process as an *execution tree* of type $T ::= \mathsf{N}\ L\ T \mid \mathsf{S}\ L\ T\ T \mid \mathsf{L}\ P\Gamma^*$. A tree of the form $\mathsf{N}\ l\ t$ models an execution that performs the non-spawning step l first, followed by the execution described by t. A tree of the form $\mathsf{S}\ l\ t_s\ t$ models an execution that performs the spawning step l first, followed by the execution of the spawned process described by t_s and the remaining execution of the spawning process described by t. A node of the form $\mathsf{L}\ pw$ indicates that the process makes no more steps and that its final configuration is pw. The annotation of the reached configuration at the leafs of the execution tree increases expressiveness of regular sets of execution trees, e.g. one can characterize execution trees that reach certain control states. The distinction between spawned and spawning tree at S-nodes allows for keeping track of which steps belong to which process, e.g. when tracking the acquired locks of a process, as done in [7].

[1] Note that the regular sets are not required to only model valid configurations, thus we could also use the set $C = (S^* p\gamma S^* p'\gamma' S^*) \mid (S^* p'\gamma' S^* p\gamma S^*)$ with $S = P \cup \Gamma$. It has a simpler automata but does not satisfy $C \subseteq \mathsf{Conf}$.

The relation $\Longrightarrow \subseteq P\Gamma^* \times T \times \mathsf{Conf}$ characterizes the execution trees starting at a single process. It is defined as the least solution of the following constraints:

$$
\begin{array}{lll}
[\text{leaf}] & qw \overset{\mathsf{L}\ qw}{\Longrightarrow} qw & \\
[\text{nospawn}] & q\gamma r \overset{\mathsf{N}\ l\ t}{\Longrightarrow} c' & \text{if } q\gamma \overset{l}{\hookrightarrow} q'w \in \Delta_N \ \wedge\ q'wr \overset{t}{\Longrightarrow} c' \\
[\text{spawn}] & q\gamma r \overset{\mathsf{S}\ l\ t_s\ t}{\Longrightarrow} c_s c' & \text{if } q\gamma \overset{l}{\hookrightarrow} q'w \rhd q_s w_s \in \Delta_S \\
& & \wedge\ q_s w_s \overset{t_s}{\Longrightarrow} c_s \ \wedge\ q'wr \overset{t}{\Longrightarrow} c'
\end{array}
$$

In order to relate the tree semantics to the interleaving semantics, we define a scheduler that maps execution trees to compatible sequences of rules. From the ordering point of view, the scheduler maps the steps ordered by the execution tree to the set of its topological sorts. The scheduler is modeled as a labeled transition system over lists of execution trees. A step replaces the root node of a tree in the list by its successors. Formally, the scheduler $\rightsquigarrow \subseteq T^* \times L \times T^*$ is the least relation satisfying the following constraints:

$$
\begin{array}{ll}
[\text{nospawn}] & h_1(\mathsf{N}\ l\ t)h_2 \overset{l}{\rightsquigarrow} h_1 t h_2 \\
[\text{spawn}] & h_1(\mathsf{S}\ l\ t_s\ t)h_2 \overset{l}{\rightsquigarrow} h_1 t_s t h_2
\end{array}
$$

We call $\mathsf{sched}(t) := \{\bar{l} \in L^* \mid \exists h' \in (\mathsf{L}\ P\Gamma^*)^*.\ [t] \overset{\bar{l}}{\rightsquigarrow}{}^* h'\}$ the set of *schedules* of a tree $t \in T$, where $(\mathsf{L}\ P\Gamma^*)^*$ is the set of all lists of L-nodes, and $\rightsquigarrow^*$ is the reflexive, transitive closure of the scheduler $\rightsquigarrow$. Notice that this definition of the scheduler corresponds to the well-known topological sorting algorithm that iteratively removes minimal elements (in this case root nodes of trees in the list) until there are no more minimal elements left. It can be shown by induction[2] that every execution of the interleaving semantics is a schedule of an execution of the tree semantics and vice versa:

Theorem 4. *Let $p \in P$, $w \in \Gamma^*$, $c' \in \mathsf{Conf}$ and $\bar{l} \in L^*$, then $pw \overset{\bar{l}}{\rightarrow}{}^* c'$ if and only if there is an execution tree $t \in T$ with $pw \overset{t}{\Longrightarrow} c'$ and $\bar{l} \in \mathsf{sched}(t)$.*

Moreover, as trees are acyclic and thus any tree has at least one topological sort, any execution tree has at least one schedule:

Lemma 5. $\forall t \in T.\ \mathsf{sched}(t) \neq \emptyset$

Reached Configuration. The execution tree already determines the configuration that is reached by the execution, as the reached configuration is annotated at the leafs of the execution tree. We define the function $\mathsf{conf} : T \rightarrow \mathsf{Conf}$ that returns the configuration reached by an execution tree:

$$
\begin{aligned}
\mathsf{conf}(\mathsf{L}\ pw) &= pw \\
\mathsf{conf}(\mathsf{N}\ l\ t) &= \mathsf{conf}(t) \\
\mathsf{conf}(\mathsf{S}\ l\ t_s\ t) &= \mathsf{conf}(t_s)\mathsf{conf}(t)
\end{aligned}
$$

[2] To get the induction through, one has to generalize the $\overset{\cdot}{\Longrightarrow}$-relation to more than one initial process and lists of execution trees, as done in [7, 6].

It is straightforward to show that $\mathsf{conf}(t)$ really returns the configuration reached by any execution of t:

Lemma 6. $\forall pw \in P\Gamma^*,\ t \in T, c' \in \mathsf{Conf}.\ pw \overset{t}{\Longrightarrow} c' \implies c' = \mathsf{conf}(t)$

3 Regular Execution Trees

An execution tree makes the structure of process creation explicitely visible. However, the structure of procedure calls and matching returns is not explicitely visible in the structure of the execution tree. The information whether a procedure call returns and where the matching return node is, can only be obtained by inspecting the execution of the process and searching for a matching return node. In this section, we develop another tree-based representation of executions, so called *regular execution trees*. The structure of those execution trees makes visible both, process creation and procedure calls. For the remainder of this paper, we will use the term *standard execution tree* to refer to the execution trees introduced above. Regular execution trees have the advantage that the set of executions starting at a configuration of the form $p\gamma$ (in particular $p_0\gamma_0$) can be described as a regular set, and it is straightforward to construct a tree-automaton for this set from the rules of the DPN. Thus, we can use standard tree-automata operations like intersection and emptiness check for the analysis. We distinguish between the set R_r of returning execution trees and the set R_n of non-returning execution trees. A returning execution tree returns from the current procedure, while a non-returning execution tree does not. Formally, regular execution trees have the type:

$$R_r ::= \mathsf{ret}\ L\ P \mid \mathsf{base}\ L\ R_r \mid \mathsf{callr}\ L\ R_r\ R_r \mid \mathsf{spawn}\ L\ R_n\ R_r$$
$$R_n ::= \mathsf{nil}\ P\Gamma \mid \mathsf{base}\ L\ R_n \mid \mathsf{callr}\ L\ R_r\ R_n \mid \mathsf{calln}\ L\ R_n\ \Gamma \mid \mathsf{spawn}\ L\ R_n\ R_n$$

Moreover, we define the set R of all regular execution trees by $R := R_r \cup R_n$. Intuitively, a $\mathsf{nil}\ p\gamma$-node represents the end of a process's execution. The process ends in state p with top-of-stack γ. A $\mathsf{ret}\ l\ p$-node represents a procedure return. The return step is labeled with l, and the return state is p. A $\mathsf{base}\ l\ \tau$-node represents a basic-step with label l. After the basic step, the execution continues with τ. A $\mathsf{callr}\ l\ \tau_c\ \tau$-node represents a returning procedure call labeled with l. The execution of the procedure is described by τ_c and the execution after the procedure has returned is described by τ. A $\mathsf{calln}\ l\ \tau_c\ \gamma$-node represents a procedure call that does not return. τ_c represents the execution of the procedure, and γ is the return address of the procedure, that will – as the procedure does not return – remain on the stack for the rest of the execution. A $\mathsf{spawn}\ l\ \tau_s\ \tau$-node represents a process creation step, where τ_s is the execution of the created process and τ is the remaining execution of the creating process. Note that, due to Assumption 2, a (reachable) spawned process does not return from its initial procedure, and thus we have $\tau_s \in R_n$.

In order to map from a regular execution tree to a standard execution tree, we need to define a concatenation operation that glues together two standard

execution trees, by replacing the rightmost leaf of the first tree by the second tree. We define the operation $\cdot;\cdot : T \times T \to T$ recursively over the structure of the first tree:

$$\begin{aligned}
&(\mathsf{L}\ pw); t' = t' \\
&(\mathsf{N}\ l\ t); t' = \mathsf{N}\ l\ (t; t') \\
&(\mathsf{S}\ l\ t_s\ t); t' = \mathsf{S}\ l\ t_s\ (t; t')
\end{aligned}$$

In order to define the function $\alpha : R \to T$ that maps from regular execution trees to standard execution trees, we first define the auxiliary function $\alpha' : R \times \Gamma^* \to T$ that maps a regular execution tree and some stack to a standard execution tree. Intuitively, the stack contains the symbols that will not be popped during the rest of the execution.

$$\begin{aligned}
&\alpha'(\mathsf{nil}\ p\gamma, s) = \mathsf{L}p(\gamma s) \\
&\alpha'(\mathsf{ret}\ l\ p, s) = \mathsf{N}\ l\ (\mathsf{L}\ ps) \\
&\alpha'(\mathsf{base}\ l\ \tau, s) = \mathsf{N}\ l\ (\alpha'(\tau), s) \\
&\alpha'(\mathsf{callr}\ l\ \tau_c\ \tau, s) = \mathsf{N}\ l\ (\alpha'(\tau_c, \varepsilon); \alpha'(\tau, s)) \\
&\alpha'(\mathsf{calln}\ l\ \tau_c \gamma, s) = \mathsf{N}\ l\ \alpha'(\tau_c, \gamma s) \\
&\alpha'(\mathsf{spawn}\ l\ \tau_s\ \tau, s) = \mathsf{S}\ l\ \alpha'(\tau_s, \varepsilon)\ \alpha'(\tau, s)
\end{aligned}$$

The function α is then defined as $\alpha(\tau) := \alpha'(\tau, \varepsilon)$. We overload $\alpha : 2^R \to 2^T$ for sets of trees by element-wise function application, i.e. $\alpha(X) = \{\alpha(\tau) \mid \tau \in X\}$.

We now characterize the regular execution trees of a DPN by the least fixed point of a constraint system. The constraint system contains variables of the form $N[p, \gamma] \subseteq R_n$ and $R[p, \gamma, q] \subseteq R_r$ for $p, q \in P$ and $\gamma \in \Gamma$. Intuitively, $N[p, \gamma]$ contains the set of non-returning execution trees starting at configuration $p\gamma$ and $R[p, \gamma, q]$ contains the set of execution trees starting at configuration $p\gamma$ and returning with state q:

$$\begin{array}{lll}
\quad [\text{n-nil}] & \mathsf{nil}\ p\gamma \in N[p,\gamma] & \\
\multicolumn{3}{l}{\text{for } p\gamma \overset{l}{\hookrightarrow} p' \in \Delta :} \\
\quad [\text{r-ret}] & \mathsf{ret}\ l\ p' \in R[p,\gamma,p'] & \\
\multicolumn{3}{l}{\text{for } p\gamma \overset{l}{\hookrightarrow} p'\gamma' \in \Delta,\ \tilde{p} \in P :} \\
\quad [\text{n-base}] & \mathsf{base}\ l\ \tau \in N[p,\gamma] & \Leftarrow \tau \in N[p',\gamma'] \\
\quad [\text{r-base}] & \mathsf{base}\ l\ \tau \in R[p,\gamma,\tilde{p}] & \Leftarrow \tau \in R[p',\gamma',\tilde{p}] \\
\multicolumn{3}{l}{\text{for } p\gamma \overset{l}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta,\ \tilde{p},\hat{p} \in P :} \\
\quad [\text{n-calln}] & \mathsf{calln}\ l\ \tau\ \gamma_2 \in N[p,\gamma] & \Leftarrow \tau \in N[p',\gamma_1] \\
\quad [\text{n-callr}] & \mathsf{callr}\ l\ \tau_c\ \tau \in N[p,\gamma] & \Leftarrow \tau_c \in R[p',\gamma_1,\hat{p}] \wedge \tau \in N[\hat{p},\gamma_2] \\
\quad [\text{r-callr}] & \mathsf{callr}\ l\ \tau_c\ \tau \in R[p,\gamma,\tilde{p}] & \Leftarrow \tau_c \in R[p',\gamma_1,\hat{p}] \wedge \tau \in R[\hat{p},\gamma_2,\tilde{p}] \\
\multicolumn{3}{l}{\text{for } p\gamma \overset{l}{\hookrightarrow} p'\gamma' \rhd p_s\gamma_s \in \Delta,\ \tilde{p} \in P :} \\
\quad [\text{n-spawn}] & \mathsf{spawn}\ l\ \tau_s\ \tau \in N[p,\gamma] & \Leftarrow \tau_s \in N[p_s,\gamma_s] \wedge \tau \in N[p',\gamma'] \\
\quad [\text{r-spawn}] & \mathsf{spawn}\ l\ \tau_s\ \tau \in R[p,\gamma,\tilde{p}] & \Leftarrow \tau_s \in N[p_s,\gamma_s] \wedge \tau \in R[p',\gamma',\tilde{p}]
\end{array}$$

In the remainder, we use $N[p, \gamma]$ and $R[p, \gamma, q]$ to refer to the least solution of this constraint system. Note that these constraints also define rules of a tree automaton over states $\{N[p, \gamma] \mid p \in P, \gamma \in \Gamma\} \cup \{R[p, \gamma, q] \mid p, q \in P, \gamma \in \Gamma\}$.

Thus, the sets $N[p,\gamma]$ and $R[p,\gamma,q]$ are tree regular sets. We will use this tree-automata view and the closedness properties of regular tree languages in order to decide whether a DPN has some execution that satisfies a given tree-regular property $\Phi \subseteq R$, as this is equivalent to $N[p_0,\gamma_0] \cap \Phi \neq \emptyset$ which can be decided by standard tree automata operations.

First, we justify the regular execution tree semantics defined by the constraint system w.r.t. the standard execution tree semantics:

Theorem 7. *Let $p, p' \in P$ and $\gamma \in \Gamma$. Then we have:*

$$\text{a)}\quad \{t \mid \exists c' \in (P\Gamma^+)^+.\ p\gamma \overset{t}{\Longrightarrow} c'\} \;=\; \alpha(N[p,\gamma])$$
$$\text{b)}\ \{t \mid \exists c_s \in (P\Gamma^+)^*.\ p\gamma \overset{t}{\Longrightarrow} c_s p'\} \;=\; \alpha(R[p,\gamma,p'])$$

Intuitively, Theorem 7a states that there is a standard execution t, starting at configuration $p\gamma$ and ending in a configuration with no empty stack ($c' \in (P\Gamma^+)^+$), if and only if there is a regular execution tree $\tau \in N[p,\gamma]$ that matches the standard execution tree t, i.e. ($\alpha(\tau) = t$). Theorem 7b makes the analog statement for returning executions.

Reached Configuration. We now extract the configuration reached by a regular execution tree: We overload the function $\mathsf{conf} : R \to \mathsf{Conf}$:

$$\begin{aligned}
&\mathsf{conf}(\mathsf{nil}\ p\ \gamma) = p\gamma\\
&\mathsf{conf}(\mathsf{ret}\ l\ p) = \varepsilon\\
&\mathsf{conf}(\mathsf{base}\ l\ \tau) = \mathsf{conf}(\tau)\\
&\mathsf{conf}(\mathsf{callr}\ l\ \tau_c\ \tau) = \mathsf{conf}(\tau_c)\mathsf{conf}(\tau)\\
&\mathsf{conf}(\mathsf{calln}\ l\ \tau_c\ \gamma) = \mathsf{conf}(\tau_c)\gamma\\
&\mathsf{conf}(\mathsf{spawn}\ l\ \tau_s\ \tau) = \mathsf{conf}(\tau_s)\mathsf{conf}(\tau)
\end{aligned}$$

It is straightforward to show that, for non-returning execution trees, the configuration computed by conf matches the configuration reached by the corresponding standard execution tree:

Lemma 8. $\forall \tau \in R_n.\ \mathsf{conf}(\tau) = \mathsf{conf}(\alpha(\tau))$

In the rest of this paper, we want to characterize the set of regular execution trees that reach a configuration that is in a given regular set of configurations. For this purpose, we regard a finite state machine (FSM) $F = (Q, q_0, Q_F, \delta)$ with states Q, initial state $q_0 \in Q$, final states $Q_F \subseteq Q$ and transition relation $\delta \subseteq Q \times (P \cup \Gamma) \times Q$. Let δ^* be the reflexive, transitive closure of δ. The language of F is defined by $L(F) := \{c \in (P \cup \Gamma)^* \mid \exists q' \in Q_F.\ (q_0, c, q') \in \delta^*\}$. We regard a constraint system over variables $T[q,q'] \subseteq R$ for $q, q' \in Q$. Intuitively, $T[q,q']$ contains all regular execution trees whose reached configuration drives the FSM from state q to state q':

$$\begin{array}{lll}
[\text{t-nil}] & \mathsf{nil}\ p\gamma \in T[q,q'] & \Leftarrow (q, p\gamma, q') \in \delta^*\\
[\text{t-ret}] & \mathsf{ret}\ l\ p \in T[q,q] & \Leftarrow q \in Q\\
[\text{t-base}] & \mathsf{base}\ l\ \tau \in T[q,q'] & \Leftarrow \tau \in T[q,q']\\
[\text{t-callr}] & \mathsf{callr}\ l\ \tau_c\ \tau \in T[q,q'] & \Leftarrow \exists \hat{q}.\ \tau_c \in T[q,\hat{q}] \wedge \tau \in T[\hat{q},q']\\
[\text{t-calln}] & \mathsf{calln}\ l\ \tau_c\ \gamma \in T[q,q'] & \Leftarrow \exists \hat{q}.\ \tau_c \in T[q,\hat{q}] \wedge (\hat{q},\gamma,q') \in \delta\\
[\text{t-spawn}] & \mathsf{spawn}\ l\ \tau_s\ \tau \in T[q,q'] & \Leftarrow \exists \hat{q}.\ \tau_s \in T[q,\hat{q}] \wedge \tau \in T[\hat{q},q']
\end{array}$$

In the rest of this paper we use $T[q,q']$ to refer to the least solution of this constraint system. Note that this constraint system also defines the rules of a tree automaton, and thus the sets $T[q,q']$ are regular. It is straightforward to show that $T[q,q']$ contains exactly those regular execution trees whose configuration transfers the FSM from state q to state q':

Theorem 9. $\forall \tau \in R,\ q, q' \in Q.\ \tau \in T[q,q'] \ \Leftrightarrow\ (q, \mathsf{conf}(\tau), q') \in \delta^*$

By combining the previous results of this paper, we can now decide whether some configuration from the regular set of configurations $L(F)$ is reachable from the initial configuration $p_0\gamma_0$:

Theorem 10 (Main Result).

$$(\exists \bar{l} \in L^*,\ c' \in L(F).\ p_0\gamma_0 \xrightarrow{\bar{l}}{}^* c') \ \Leftrightarrow\ N[p_0,\gamma_0] \cap \bigcup\{T[q_0,q'] \mid q' \in Q_F\} \neq \emptyset$$

Note that both $N[p_0,\gamma_0]$ and $\bigcup\{T[q_0,q'] \mid q' \in Q_F\}$ are regular tree languages, and the rules of the corresponding tree automata can be constructed from the scheme given by the constraint systems for N, R and T. Hence, the right hand side of the equivalence can be decided using standard tree automata algorithms for intersection and emptiness check.

Proof. For the $\Rightarrow$-direction assume there is an sequential execution $\bar{l} \in L^*$ and a configuration $c' \in L(F)$ with $p_0\gamma_0 \xrightarrow{\bar{l}}{}^* c'$. With Theorem 4, we obtain an standard execution tree t with $p_0\gamma_0 \overset{t}{\Longrightarrow} c'$. From Assumption 2 and Lemma 3 we have $c' \in (P\Gamma^+)^+$, hence we can use Theorem 7a to obtain a regular execution tree $\tau \in N[p_0,\gamma_0]$ with $\alpha(\tau) = t$.

Moreover, with Lemma 6, we have $\mathsf{conf}(t) = c'$, and with Lemma 8 we get $\mathsf{conf}(\tau) = c'$. With $c' \in L(F)$ we obtain a state $q' \in Q_F$ with $(q_0, \mathsf{conf}(\tau), q') \in \delta^*$. With Theorem 9, we have $\tau \in T[q_0,q']$ and hence $\tau \in \bigcup\{T[q_0,q'] \mid q' \in Q_F\}$. Ultimately, we get $N[p_0,\gamma_0] \cap \bigcup\{T[q_0,q'] \mid q' \in Q_F\} \neq \emptyset$.

The proof of the $\Leftarrow$-direction is analog. Only in order to apply Theorem 4, we need the additional fact that every standard execution tree has at least one schedule (Lemma 5). □

4 Conclusion

We presented a tree-based semantics for DPNs. The set of executions of the DPN are a regular set, and a tree automata for this set can efficiently be derived from the DPN rules. The semantics is justified w.r.t. the true-concurrency semantics presented in [7] and thus, indirectly, w.r.t the original interleaving semantics of DPNs [2]. By using standard tree automata techniques, we are able to decide tree regular properties of DPN executions. To show the usefulness of tree-regular properties, we showed that reaching a configuration from a given regular set is a tree-regular property.

All the results presented in this paper have been formalized in the interactive theorem prover Isabelle/HOL [9]. The proofs related to DPNs and standard execution trees have been published as a technical report [6]. The proofs related to regular execution trees are still unpublished.

Future Work Currently, we have only modeled one specific property – execution trees reaching a configuration from a regular set – as a tree-automaton and justified it by hand. However, there is strong indication that the function α, that maps from regular execution trees to standard execution trees, can be written as a macro tree transducer. Then, for every regular set $S \subseteq T$ of standard execution trees, the set $\alpha^{-1}(T) \subseteq R$ of corresponding regular execution trees is also regular, and an automaton for $\alpha^{-1}(T) \subseteq R$ can be constructed automatically. This would allow us to automatically transfer other useful properties of standard execution trees to our new analysis technique, e.g. the analysis of reachability w.r.t. nested locks presented in [7].

By iterating pre^*_M-computations, one can check for executions that reach a sequence of intermediate configurations from given regular sets. This can be used for, e.g., bounded model checking of DPNs with shared memory [1]. In order to achieve the same with our tree-automata based techniques, we have to specify tree automata for execution trees that reach a sequence of intermediate configurations, each from a given regular set. Ideally, we would develop a method to automatically derive these tree automata from regular properties that are separately specified for each execution between two intermediate configurations.

Another direction of future work is to extend the power of the analyzed model. While, in this paper, we only presented an analysis for DPNs, we are pretty sure that our results carry over to the strictly more powerful CDPNs [2], where rules may be constrained by properties of the state of the processes that have been spawned by the process that executes the rule. This allows, e.g., to model parallel procedure calls. Moreover, we could try to use other techniques, in particular Horn-Clause solvers and symbolic representations of the tree automata rules, to improve the performance of the analysis, in particular for large DPNs or properties where the automaton has many states but a possible short symbolic description.

Moreover, we have to compare our analysis techniques with the existing techniques for pre^*_M-computations. One aspect of this comparison is to compare the theoretical worst case complexities of the two methods. Another aspect would be an experimental evaluation of the runtimes for typical problems. We currently have prototype implementations for both techniques, but no experimental results yet, nor any collection of ,,typical problems".

Acknowledgment. We thank Helmut Seidl for helpful discussion about tree automata and Markus Müller-Olm and Alexander Wenner for inspiring discussion about DPNs. Actually, the idea of regular execution trees evolved during a meeting with Helmut, Markus and Alexander.

References

1. A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *Proc. of FSTTCS'05*, volume 3821 of *LNCS*, pages 348–359. Springer, 2005.

2. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.
3. V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *Proc. of LICS 2006*, pages 101–110. IEEE Computer Society, 2006.
4. V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL*, pages 303–314, 2007.
5. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV 2005*, volume 3576 of *LNCS*. Springer, 2005.
6. P. Lammich. Isabelle formalization of hedge-constrained pre* and DPNs with locks. Available from `http://cs.uni-muenster.de/sev/publications/`. Technical Report.
7. P. Lammich, M. Müller-Olm, and A. Wenner. Predecessor sets of dynamic pushdown networks with tree-regular constraints. In *Computer Aided Verification*, volume 5643 of *LNCS*, 2009.
8. M. Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theor. Comput. Sci.*, 311(1-3):325–388, 2004.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
10. H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.

MultiMediaC# - QoS-Aware Programming with C#

Oliver Lampl, Laszlo Böszörmenyi

Institute of Information Technology, University of Klagenfurt, Austria
olampl@edu.uni-klu.ac.at, laszlo@itec.uni-klu.ac.at

Abstract. Providing QoS-awareness and adaptive behavior in multimedia applications is cumbersome and errorprone. Many different frameworks and even additional languages exist which try to support QoS-aware application development. In many cases this is done by shifting away QoS processing facilities from the prgrammer and hiding them inside resource layers or other middle-ware. Instead, we integrate QoS directly into the common programming language C#.
In this paper we introduce the specification of MMC# as an extension to the C# programming lanugage focusing on adaptive, QoS-aware programming. The new language features provide a solution to the three common problems in the field of QoS processing: (1) constraint declaration, (2) constraint monitoring and (3) providing adaptive behavior in multimedia applications. Furthermore, declarative definition of QoS requirements directly within the programming language allows to apply semantic constraint analysis - partly at compile-time, partly run-time - to check the correctness of specified requirements and further provide optimizations by automatically removing not required constraints. Last but not least, this paper describes the structure of the Mono compiler which has been the basis for development. We illustrate how the compiler has been modified to support these novel language extensions and the semantic analysis of QoS constraints.

1 Introduction

Up to the present, the software engineering community has largely neglected the problems surrounding the development of multimedia applications that dynamically adapt their behavior to both the required Quality of Service (QoS) and to available resources. QoS and adaptation of course have been recognized as important issues and have been subsequently discussed in the relevant literature, but these discussions have usually restricted themselves to a limited context and rarely concentrated on the design issues of QoS aware, adaptive applications.

Many different frameworks exist which promise support for QoS handling and adaptivity which is achieved by defining QoS contracts within separate specification languages. Furthermore, a lot of multimedia frameworks and middleware systems have been implemented which claim to provide QoS handling. However, their QoS capabilities are often limited to a small context or even completely hidden to the programmer.

Currently, there is no support for QoS-aware programming within common general purpose programming languages like C# or Java. We introduce an extension to the programming language C#, called MMC#, which focuses on adaptive QoS-aware programming. MMC# extends current C# programming capabilities by adding the following features:

- Constraint declaration - A declarative syntax is used to provide QoS constraint specification.
- Constraint monitoring - Timed data types utilize the $n + 1^{st}$ dimension as representation of the time and provide historization with the help of special assignment operations. These timed assignments apply automatic time recording and QoS constraint evaluation.
- Adaptivity - In case of QoS violations exception are thrown, which enable adaptive programming to be as easy as any other kind of exception handling.

Furthermore, we apply semantic analysis on declarative constraint specification to provide their correctness and even optimize their representation. This analysis is splitted into two parts:

- Compile-time Analysis - The compiler analyzes the constraints and provides correctness checks and optimizations for constant declarations. In case of dynamic values it generates code which is executed during constraint initialization.
- Run-time Analysis - Constraint declaration can use dynamic elements like parameters or method calls which are not available during compile-time. These values are analyzed during run-time and in case they lead to an incorrect constraint definition, an exception is thrown.

MMC# and the semantic analysis have been implemented within the Mono C# compiler. The Mono framework is an open-source implementation of the .NET framework and provides portability of .NET applications to other platforms than Windows. We briefly describe the structure of the compiler and show how the presented extensions have been integrated.

2 MMC# Specification

Keywords

```
constraint
stream
```

Types

```
type:
        timed-type
        value-type
        reference-type
        type-parameter

timed-type:
        non-timed-type [ ~ constraint-reference_opt ]

non-timed-type:
        value-type
        reference-type
```

Assignments

```
assignment:
        unary-expression assignment-operator expression
        unary-expression ~: expression
        unary-expression :~ unary-expression
```

Methods

```
parameter-modifier:
        ref
        out
        stream
```

```
argument:
        expression
        ref variable-reference
        out variable-reference
        stream expression
```

Classes

```
class-member-declaration:
        constant-declaration
        parameterizable-constraint-declaration
        field-declaration
        method-declaration
        property-declaration
        event-declaration
        indexer-declaration
        operator-declaration
        constructor-declaration
        finalizer-declaration
        static-constructor-declaration
        type-declaration

parameterizable-constraint-declaration:
        constraint-modifiers_opt constraint identifier
(formal-parameter-list_opt) => constraint-declarations ;

constraint-modifiers:
        constraint-modifier
        constraint-modifiers constraint-modifier

constraint-modifier:
        private
        protected
        public
        static

constraint-declarations:
        or-constraint-declaration

or-constraint-declaration:
        and-constraint-declaration
        or-constraint-declaration ||
and-constraint-declaration

and-constraint-declaration:
        constraint-declaration
        and-constraint-declaration &&
or-constraint-declaration

constraint-declaration:
        constraint-declaration
        and-constraint-declaration &&
or-constraint-declaration
```

Constraint declaration

```
constraint-declaration:
        @ identifier { constraint-condition-with-exception
}

constraint-condition-with-exception:
        or-constraint-condition
        or-constraint-condition : throw-statement

or-constraint-condition:
        and-constraint-condition
        or-constraint-condition || and-constraint-condition

and-constraint-condition:
        constraint-condition
        and-constraint-condition && or-constraint-condition

constraint-condition:
        qos-expression
        qos-expression qos-conditional-operator
qos-expression

qos-conditional-operator:
        <
        <=
        ==
        >
        >=

qos-expression:
        qos-primary-expression
        qos-expression qos-math-operator qos-expression

qos-math-operator:
        +
        -
        *
        /
        %

qos-primary-expression:
        primary-expression
```

2.1 Types

The type definition[1] is extended to support timed-types. Timed-types are value-types and similar to array-types.

A timed datatype in this context also called quality-aware data type is defined similarly to an array type. Time is the $n+1^{st}$ dimension, which can be added to any data type. Quality-aware data types are associated with a history which records events. These events are triggered by the timed (streaming) assignment statement (see section 2.2) and the timed (streaming) parameter passing mode (see section 2.3).

Quality-aware variables are type-compatible with any variable compatible to their base type. During type-checking the additional dimension is omitted. To allow QoS checking, constraints are associated to timed data types. Such constraints are of type `System.IQoSObject` and are either added statically during declaration or assigned dynamically via the `new` statement.

[1] This block refers to section 11 page 107 and following of the C# Standard [1].

2.2 Assignment operators

Two new assignment operators[2] are added. The so called timed-assignment or streaming-assignment assigns a value from or to a timed-variable. This operation is done under QoS control. It advises the compiler to add QoS monitoring code.

Each time the assignment is executed the timed-variable records an event and the assigned constraint is evaluated. If the evaluation fails, an exception is thrown. This behavior is explained in section 2.5 in detail.

A write-timed-assignment requires the left part of the assignment to be a timed-type. Accordingly, a read-timed-assignment requires a timed type on the right side.

Statically assigned constraints are initalized when declaring a variable. Such constraints cannot be changed over the life-time of the variables they are associated. Instead, dynamic constraint assignments are applied using the `new` operator.

In multimedia streaming applications two different kinds of value assignments exist. The first is the usual management operation and does not need any constraint-checking or quality requirements. This operation is expressed by the "normal" assignment operation (`=`) which is executed best effort. In general, no specific timing requirements exist for management operations. In constrast, a streaming operation exists which implements features like video playback. This streaming behavior is achieved by the timed (streaming) assignment statements and requires QoS-aware behavior.

2.3 Methods

A further parameter passing mode[3] is added. `Stream` parameters are passed by value, but with constraint evaluation. Accordingly, a streamed argument is denoted with the keyword `stream`. Timed parameters require a timed data type within the declaration of the parameter list. The QoS constraint is attached when the method is called for the first time. In case of static methods only statically assigned constraints can be used.

2.4 Classes

The class member declaration[4] is extended to provide parameterizable constraints. Parameterizable constraints allow the definition of reusable constraints by specifying constraint templates as class members via the keyword `constraint`.

Constraints can be combined by logical operators `&&` and `||`. This allows building complex constraints out of simple ones, e.g combining throughput and jitter constraints. In case of a QoS constraint violation, an exception is thrown. The semantic of logical concatenation of using the operator `&&` is that every

[2] This block refers to Section 14.14 page 218 and following of the C# Standard [1].
[3] This block refers to Section 17.5.1 page 287 and following of the C# Standard [1].
[4] This block refers to Section 17.2 page 269 and following of the C# Standard [1].

constraint needs to be met. In contrast, when using the operator `||`, one single constraint satisfaction is enough to fulfill the whole constraint.

Parameterizable constraints allow QoS constraints to be reused. Such constraints are declared in the same way as other class members and make use of the same accessibility features. Furthermore, they allow the definition of parameters which can then be used inside the constraint. During constraint instantiation actual parameters are mapped to the formal parameters in the constraint definition.

2.5 Constraint declaration

Constraints are declared similarly to the mathematical forall expression ($\forall$). The identifier symbolizes a control variable which is incremented each time the constraint is evaluated. The initial value is 0, which means that no timed assignment was executed. The current value of the control variable represents the current event. This identifier can be used to access the timing history of a timed-type. Only past events can be accessed.

The constraint condition evaluates to a boolean value. If the resulting value is `false`, the constraint is violated and an exception is thrown.

A throw statement[5] can be optionally used to throw user defined exceptions in case of QoS violations. If no exception is specified, a `System.QoSException` is thrown.

Constraint conditions can be combined by logical operators `&&` and `||` which allows building complex logical expressions. Such complex conditions are evaluated as one single block.

A constraint condition is required to be of a type that can be implicitly converted to a boolean value. It allows a set of comparison operators to be used.

A `qos-expression` is used to apply mathematical operations within parts of constraints. It is clearly defined which operations are allowed to reduce complexity of constraint declaration.

Primary expressions[6] are the simplest forms of expressions. They allow e.g. variable access, paranthesis and array element access. This array element access is applied to access the timing history of a timed variable within a constraint declaration.

A constraint is represented by the implementation of the *System.IQoSObject* interface. This interface defines a method in which the history of quality-aware data types is checked against certain quality conditions. If the condition evaluates to `false`, an exception is thrown (`System.QoSException`). A constraint class can be implemented manually or generated by the compiler using the explained declarative specification syntax.

Furthermore, a constraint is only valid if there is enough information in the history. If the history does not contain enough events, the declared constraint cannot be checked and is always evaluated to true. The event counter which is declared using the symbol @ is used to access the current history entry.

[5] This block refers to Section 15.9.5 page 243 of the C# Standard.
[6] This block refers to Section 14.5 page 165 of the C# Standard.

3 Semantic Constraint Analysis

In some cases, complex constraint definitions may be hard to understand and mistakes can occur. Correctness checks executed by the compiler and by compiler generated code at run-time help the application developer to define correct constraints by applying semantic analysis and also provides optimizations by removing unnecessary constraints.

The declarative syntax allows automatic validation of syntax and semantic behavior at compile time. Syntactic correctness is achieved by correct grammar definition. The semantic analysis is divided into two parts:

1. Compile-time analysis
2. Run-time analysis

The first part is executed during compile-time and evaluates constant expressions which are subsequently compared to each other. Depending on the result of the comparison, the compiler optimizes the constraint or generates a compile-time error message or warning. If the constraint contains dynamic elements like parameters or method calls, an additional code is generated to carry out the check during run-time.

The second part of the compiler generated semantic analysis is performed at run-time, during constraint instantiation. All parameter values are evaluated against the same criteria as constant expressions during compile-time. In case of inconsistent constraint definitions, an exception is thrown. This can be compared to array bound checking [3] which is used in mainstream programming languages like C# or Java [4].

Although semantic analysis examines constraints at compile-time and during instantiation, some uncertainty of correctness remains when using user defined methods. Adding method calls allows constraints to be extensible for any required behavior, but they can also harm a constraint definition. A method call may last too long and the constraint check itself may take longer than the action being monitored. Furthermore, such method calls may lead to erroneous behavior which is eventually not captured by exception handling code. The current implementation of semantic analysis allows user defined methods, but does not analyze them in detail, except methods defined in the system class `QoS.Units`.

After the constraint analysis has been completed, a matrix with relations between constraints and parts of constraints is available. These relations help the compiler making decisions about possible optimizations. A relation between two constraints in the matrix can be expressed by one of the following types:

– **Weaker** (W) - A weaker constraint is logically implied by a stronger one.
– **Equality** (E) - Two constraints express an equivalent behavior.
– **Stronger** (S) - It is the inverse relation to the *Weaker* relation.
– **Inconsistent** (I) - Inconsistent constraints represent contradictions. Such constraints will never evaluate to true.
– **Different** ($-$) - Two constraints are different and thus both are required to express the specified requirement.

– **Dynamic Constraint** (D) - The constraint uses dynamic elements and cannot be evaluated at compile-time. Dynamic elements are method calls or class member accessses which result in varying values during run-time.

The constraint analysis process consists of the following steps:

1. Split constraint definition into conditional blocks.
2. Convert condition blocks into normalized form.
3. Compare each normalized constraint condition to each other and fill evaluation matrix.
4. Decide constraint reconstruction based on evaluation matrix to provide optimized code or warning and error messages.

3.1 Constraint Splitting

Concatenated constraints are too complex to be analyzed. Therefore, the whole constraint is split into its conditional parts. Time independent constraint parts are omitted. These parts are not covered by the constraint analysis.

An example is used to illustrate the constraint analysis. It analyzes the following QoS requirements which are expressed in Listing 1.1:

– A frame rate of 25 frames per second should be used for playback.
– The delay between two consecutive frames should not exceed 100 milliseconds.
– The deviation of delay between two consecutive frames should not exceed 20 milliseconds with respect to the frame rate of 25 frames per second.
– A group of 5 frames needs to be processed within 500 milliseconds.

```
1 @n{frame[n] - frame[n-25] < QoS.Units.Sec(1)} &&
2 @n{frame[n] - frame[n-1] < QoS.Units.MSec(100)} &&
3 @n{   frame[n] - frame[n-1] > QoS.Units.MSec(20)
4    && frame[n] < frame[n-1] + QoS.Units.MSec(60)} &&
5 @n{frame[n] - frame[n-5] < QoS.Units.MSec(500)}
```

Listing 1.1. Declarative QoS Constraint Specifying QoS Requirements.

After constraint splitting is finished, the following conditional blocks have been identified (Table 3.1).

Part	Condition
C_1	`frame[n] - frame[n-25] < QoS.Units.Sec(1)`
C_2	`frame[n] - frame[n-1] < QoS.Units.MSec(100)`
C_{3a}	`frame[n] - frame[n-1] > QoS.Units.MSec(20)`
C_{3b}	`frame[n] < frame[n-1] + QoS.Units.MSec(60)`
C_4	`frame[n] - frame[n-5] < QoS.Units.MSec(500)`

Table 1. List of Extracted Conditions.

3.2 Constraint Condition Normalization

To make constraints comparable they need to be transformed into a normalized form (Table 3.2). All constraints are converted into one of these forms (1). If a constraint condition cannot be converted into one of these forms, it is considered to be different.

$$\begin{aligned} &\forall n, \tau(\epsilon, n) - \tau(\epsilon, n-k) < \delta \\ &\forall n, \tau(\epsilon, n) - \tau(\epsilon, n-k) = \delta \\ &\forall n, \tau(\epsilon, n) - \tau(\epsilon, n-k) > \delta \end{aligned} \tag{1}$$

Part	Condition
C_1	`frame[n] - frame[n-25] < QoS.Units.Sec(1)`
C_2	`frame[n] - frame[n-1] < QoS.Units.MSec(100)`
C_{3a}	`frame[n] - frame[n-1] > QoS.Units.MSec(20)`
C_{3b}	`frame[n] - frame[n-1] < QoS.Units.MSec(60)`
C_4	`frame[n] - frame[n-5] < QoS.Units.MSec(500)`

Table 2. List of Normalized Conditions.

3.3 Constraint Condition Comparison

Constraint conditions are compared based on the defined normalized form. The parameters k and δ are the condition parameters. They can be a constant value or defined as a parameter in case of parameterizable constraints. If they are constant, they can be used for further investigations. In case they are passed as parameters they are marked as a *dynamic constraint* (D).

If constraint parts use the same timed variables, they can be compared. If they use different timed variables, it is not possible to compare them.

$$R_{A\rightarrow B} = \begin{cases} E, & \text{if } \delta_A = \delta_B \wedge k_A = k_B \\ W, & \text{if } op_A = op_B = \begin{cases} <, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} \geqslant 0 \wedge k_A \geqslant k_B \\ =, & \text{if } \frac{\delta_A}{k_A} > \frac{\delta_B}{k_B} \\ >, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} < 0 \wedge k_A < k_B \end{cases} \\ S, & \text{if } op_A = op_B = \begin{cases} <, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} \leqslant 0 \wedge k_A \leqslant k_B \\ =, & \text{if } \frac{\delta_A}{k_A} < \frac{\delta_B}{k_B} \\ >, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} > 0 \wedge k_A > k_B \end{cases} \\ D, & \text{if } \delta_A \vee \delta_B \vee k_A \vee k_B \text{ are dynamic} \\ I, & \text{if } op_A \neq op_B \wedge \begin{cases} <>, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} < 0 \\ ><, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} > 0 \end{cases} \\ -, & \text{else} \end{cases} \tag{2}$$

Condition relations are evaluated by comparing k, δ and the comparison operator (2). The method call to the class `QoS.Units` is not important in this case, due to the fact that the method body of the static methods only contain formulas with constant values. The relation matrix is the result of the comparison of these conditions. Table 3.3 illustrates the complex relations between the single conditions.

	C_1	C_2	C_{3a}	C_{3b}	C_4
C_1	E	-	-	-	-
C_2	-	E	-	W	S
C_{3a}	-	-	E	-	-
C_{3b}	-	S	-	E	S
C_4	-	W	-	W	E

Table 3. Relation Matrix.

3.4 Constraint Reconstruction

```
1 @n{frame[n] - frame[n-25] < QoS.Units.Sec(1)}
2   &&
3 @n{frame[n] - frame[n-1] > QoS.Units.MSec(20)
4       && frame[n] < frame[n-1] + QoS.Units.MSec(60)}
```

Listing 1.2. Reconstructed QoS Constraint.

The relation matrix allows reasoning about the importance of specific conditions (Table 3.4). If the matrix (Table 3.3) contains equal conditions other than self equality (main diagonal), these conditions can be reduced to one single representative. Weaker conditions can be omitted and only stronger ones are taken into account. Different conditions are important to preserve the behavior of the whole constraint. Inconsistent constraint declarations cause the compiler to produce error messages and the compilations fails. If the constraint is modified by the compiler, warnings are produced which inform the developer about the inefficient definition of the constraints.

```
1   @n{   // delay of 30 milliseconds
2   input[n] - input[n-1] < QoS.Units.MSec(30)
3      && // frame rate of 25 frames per sec
4   input[n] - input[n-25] < QoS.Units.MSec(1000)}
```

Listing 1.3. Weaker Constraint Example.

If dynamic conditions are encountered, the compiler generates check code which is executed at run-time during constraint initialization. This check code compares passed parameter values with other parameters or constant values defined within the constraint. If the check fails an exception is thrown (`System.QoSInitializationException`).

Based on the information within the relation matrix a new constraint can be built.

Relation	Description
W	If one constraint is semantically weaker than another, the weaker one can automatically be removed. A weaker constraint is logically implied by a stronger one. Listing 1.3 presents an example of this. If the delay between two consecutive frames never exceeds 30 msec then this implies that the frame rate is below 25 fps (it is actually 33,33 fps).
E	If two constraints are equivalent, one can be removed.
S	If one constraint is semantically stronger than another, the weaker one can be removed automatically.
I	If two constraints contradict each other, a compile-time error is generated. Listing 1.4 illustrates that either the first evaluates to true and the second fails or the second evaluates to true and the first fails. Such a constraint is obviously erroneous and should be removed.
–	Two constraints are different and thus both are required to express the specified requirement. No action is required.
D	The constraint uses dynamic elements and cannot be evaluated at compile-time. Dynamic elements are method calls or class member accesses which result in varying values during run-time. Check code is generated which evaluates constraint correctness during initialization.

Table 4. Relation-based Actions.

```
1  @n{ // maximum delay of 30 milliseconds
2  input[n] - input[n-1] < QoS.Units.MSec(30)
3     && // minimum delay of 50 milliseconds
4  input[n] - input[n-1] > QoS.Units.MSec(50)}
```

Listing 1.4. Inconsistent Constraint Example.

4 Extending the Mono C# Compiler

This chapter describes briefly the structure of the monon C# compiler and how modifications are applied.

5 Mono Project

The Mono project [5, 6] is an open source implementation of the .NET platform mainly developed for UNIX, but also supports Windows and Mac OS X. Its aim is to provide a cross-platform for .NET applications to enable windows applications to run on UNIX based systems and Mono developed applications to run on Windows based systems. Mono includes the following components:

- **Common Language Infrastructure** - The CLI is a virtual machine which contains a class-loader, JIT (just-in-time) compiler and GC (garbage collector) infrastructure.
- **Class Library** - The provided class library is .NET compatible and also includes Mono-provided classes.

– **C# Compiler** - The C# compiler is the main compiler of the Mono framework. There is already support for other languages like Visual Basic, Oberon or Object Pascal.

The .NET framework uses a virtual machine like environment called the Common Language Runtime (CLR) and is intended for use with several different programming languages. The .NET platform has compilers [9] that target the virtual machine from a number of languages: Managed C++, Java Script, Eiffel, Component Pascal, just to name a few. The CLR and the Common Type System (CTS) provide a common run-time to all of these languages. Independent of their syntax and language structures the target output is the Common Intermediate Language (CIL) which allows interoperability between all CLI compatible languages. When a CIL application is executed, the code is compiled to native machine code for the appropriate architecture.

6 Structure of the Mono C# Compiler

The Mono C# compiler is one central aspect of the Mono framework. It is an ECMA-Standard [1] compliant implementation of a compiler for the C# programming language targeting the standard compliant CLI [2]. Each new version of the programming language is provided via a separate compiler executable.

– **mcs** - A C# compiler targeting .NET 1.x framework.
– **gmcs** - A C# compiler targeting .NET 2.0 and 3.5 framework.
– **smcs** - A C# compiler targeting .NET 2.1 framework including Silverlight APIs.
– **dmcs** - A C# 4.0 compatible compiler (currently under development).

The implementations of MMC# started with implementing the language extensions with mcs. Later on the development moved to gmcs.

Mono C# compiler [7] is completely written in the C# programming languages and makes heavy use of .NET API calls inlcuding `System.Reflection` and `System.Reflection.Emit`.

The compilation process is divided into the following parts:

1. **Parsing** - The compiler parses a set of source files. This is done with the JAY parser which is a port of Berkeley Yacc to Java that was later ported to C# by Miguel de Icaza. Additionally used assemblies are loaded after parsing has been completed.
2. **Type Hierarchy Processing** - After parsing has finished the type hierarchy is resolved and populated to the type system. After this step the program skeleton is complete.
3. **Code Generation** - This phase is divided into two parts. The first part is responsible to provide semantic analysis and check if the code is correct. The second one really emitts the code. After executing code generation the output is saved to the disk.

[7] provides a detailed overview about the file organization of the compiler and explains the responsibilities of each code block.

6.1 Lexical Analysis

During the first phase of the compiler execution all input is processed by the tokenizer. The tokenizer splits the input text into small parts which then are used by the JAY parser for further processing.

If the tokenizer returns a token, its location is recorded within a property which is accessible by the parser. This information is used to provide correct error messages pointing to the location of the problem including line number and column. If the location cannot be determined by the compiler, error messages pointing to line 0 are produced which represent anonymous error messages.

C# has only limited pre-processing capabilities compared to the programming language C [10]. A separate method is invoked when detecting pre-processing directives which start with the symbol #.

6.2 Syntax Analysis

The JAY parser takes the previously generated tokens and evaluates the syntax of the source program. The grammer of the parser does not exactly match the grammar provided within the standard, because the standard has been written for human beings and not for machines.

Each statement or expression identified by the parser is represented by a class. If the parser finds an appropriate token sequence, it instantiates an object of the corresponding class representing the analyzed language construct.

The output of the syntax analysis is an abstract syntax tree (AST) which contains expression trees and all statements required for code generation. It further allows semantic analysis to check the correctness of the code and evaluation of all used types.

6.3 Semantic Analysis

The abstract syntax tree is the internal representation of the provided program. It is the input for the semantic analysis. This analysis starts by resolving the interface hierarchy. Next, classes, structures, constants and enumerations follow. After processing the main building blocks of applications methods, indexers, properties, delegates and events are resolved. At the end of this process elements containing code are checked to be semantically correct.

6.4 Code Generation

The code generation is done with the help of context objects (`EmitContext`) which keep track of current namespaces, type containers and the state of code generation. The resolved abstract syntax tree is the basis for code generation. With the help of the `ILGenerator`, the program is translated into the Common Intermediate Language (CIL).

7 Embedding Language Extensions

Depending on the required enhancements, several different possibilities exist to modify the existing Mono C# compiler. The implementation of the language features presented within this paper have been applied to gmcs which is the compiler version targeting the .NET framework 2.0.

The tokenizer is based on a manual implementation and needs to be modified accordingly in case new tokens are required. For the current extensions this was made for the new assignment statements and keywords introduced within MMC#.

The parser is implemented as a JAY-parser specification which is a mixture of grammar definitions and related C# code which is executed when a given token sequence is recognized. The modifications to the grammar were directly applied within the "*cs-parser.jay*" file. During the parsing process an abstract syntax tree which consists of abstract representations of expressions and statements is generated.

There are many different possibilities of how a certain modification can be implemented. Instead of emitting code manually, the presented language features are based on modifications directly applied to the abstract syntax tree. This can be explained considering the following example. If an assignment statement is detected, an instance of the class `Assign` is created. This object contains a left-hand and a right-hand expression which are the target and the source of the assignment operation respectively.

During semantic analysis this object is resolved by identifying the types of the expressions provided and code generation can emit the appropriate code. In case of the timed assignment, the implementation could be similar and a new object could have been created which then emits the required code. Instead, the representation of the operation within the abstract syntax tree was modified to contain not only an assignment object but further objects representing method invocations to the run-time available QoS management framework. In other words, the current implementation reuses existing elements of the abstract syntax tree to provide the new functionality.

This implementation technique has the advantage to not worry about the Common Intermediate Language (CIL). Correct code generation is left to the existing implementation of the compiler. This approach allows easy modifications to the front-end of the compiler without worrying about the code generation in detail.

The use of timed data types as overlay types - this means that the additional type information is ignored when applying type compatibility checks - allows the generated CIL code to be standard conform. It further enables compiled applications to run on different .NET implementations and in addition allows QoS-aware libraries to be reused within standard .NET or Mono programs.

If timed data types had been integrated into the Common Type System (CTS), other programming languages could use the new types. This has the disadvantage that a QoS-aware .NET runtime needs to be shipped with each application instead of reusing existing installations. However, this approach was

not chosen to provide compatibilty to existing implementations which might also increase acceptance.

7.1 Applied Modifications

The following files are modified to implement the presented language enhancements for *Quality Aware Programming.*

- **assign.cs** - This file contains all classes which are required to implement an assignment operation and the required object initializer. The MMC# modification is required for the constraint assignment when initializing a time variable with a new constraint.
- **cs-parser.jay** - This is the parser definition file which reads the input source code and provides the abstract syntax tree. It contains most of the modifications which have been applied.
- **cs-tokenizer.cs** - The tokenizer is extended to accept new keywords and the timed assignment operation.
- **delegate.cs** - This file contains all classes which deal with delegates. The implementation of timed parameters requires modification to this file.
- **ecore.cs** - This file contains base classes for expressions and types. It includes an implementation of the timed data type which is implemented as an overlay datatype which can be applied to any other data type.
- **expression.cs** - All expressions are implemented within this file. Time parameter arguments are implemented as well as the cast operation which is required to ignore the timed type to provide correct type compatibility with non-timed types.
- **parameter.cs** - This file is responsible for method parameters. It contains the extensions for timed method parameters.

8 Conclusion and Future Work

This paper presents an approach of adaptive QoS-aware programming with the help of MMC#. It integrates a few novel language extensions directly within the common general purpose programming language C#. Furhtermore, we show how semantic analysis can be applied partly at compile-time and partly at runtime with the help of compiler-generated code. This analysis provides correctness checks and optimizes constraint definitions. Last but not least, the integration within the Mono framework is explained.

There are still many open questions which require further research. We need to define a set of programming patterns to show how the presented programming features should be applied in QoS-aware applications. Furthermore, a detailed analysis of system influence on QoS monitoring needs to be done. The impact of operating system scheduling and the use of garbage collection on QoS monitoring needs to be reflected in detail.

Furthermore, we want to check possibilties if compile time type safety can be extended to handle QoS negotiation within a QoS-aware multimedia framework.

All in all, this is one step to make QoS-aware programming widely available and to increase the number of adaptive applications by making their development much easier.

References

[1] C# Language Specification Standard ECMA-334, June 2005

[2] Common Language Infrastructure (CLI) Standard ECMA-335, June 2006

[3] , Nguyen,, Thi Viet Nga and Irigoin,, François: Efficient and effective array bound checking ACM Trans. Program. Lang. Syst., 2005, Vol 27, pages 527-570, New York, NY, USA

[4] Java, 2009 http://java.sun.com/

[5] Mono: Open Source .NET Development Framework, 2009 http://www.mono-project.com

[6] Mono (software), 2009 From Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Mono_(software)

[7] Miguel de Icaza: The Internals of the Mono C# Compiler, 2009 From Mono-Project Subversion

[8] Miguel de Icaza: The Mono C# Compiler, 2002 From Mono-Project, http://primates.ximian.com/ miguel/slides-europe-nov-2002/Mono_C_Sharp_Overview_1007.sxi

[9] List of CLI languages, 2009 From Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Microsoft_.NET_Languages

[10] Brian W. Kernighan and Dennis Ritchie: The C Programming Language, Second Edition Prentice-Hall, 1988, ISBN 0-13-110370-9

Technische Aspekte des erfolgreichen Testens von Software in Unternehmen

Tim A. Majchrzak

Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster
Münster, Germany
tima@wi.uni-muenster.de

Zusammenfassung Um Softwareprodukte von hoher Qualität zu erstellen ist das Testen unerlässlich. Da es sich bei Softwaretests um eine teure Aufgabe handelt, die nur schwierig zu beherrschen ist, muss ihre organisatorische Einbettung wohlüberlegt sein. Wir haben mit regionalen Unternehmen zusammengearbeitet, um ihre individuellen Stärken und Schwächen hinsichtlich der Entwicklung und insbesondere des Testens von Software kennenzulernen. In der Folge war es uns möglich, erfolgreiche Vorgehensweisen ("best practices") abzuleiten und Empfehlungen zu formulieren. In diesem Artikel wird das Projekt und die gewählte Forschungsmethodik vorgestellt. Danach werden fünf Empfehlungen vorgestellt, deren Fokus auf technischen bzw. technologischen Aspekten des Testens liegt. Es wird insbesondere auch berücksichtigt, welchen Einfluss Programmierpraktiken sowie -paradigmen bzw. die Wahl der Programmiersprache haben. Für jede Empfehlung wird erörtert, unter welchem Gegebenheiten sich ihre Umsetzung anbietet.

1 Einführung

Das Streben nach hoher Softwarequalität ist keine neue Erscheinung. Es wird bereits seit Jahrzehnten versucht, die Qualität mit Hilfe verschiedener Techniken und der Entwicklung neuer Technologien zu erhöhen. Dementsprechend ist der Begriff *Software Engineering* [1] bereits in den 1960er Jahren geprägt worden. Die *Softwarekrise* währt schon seit den 1970ern [2].

Es sind vor allem Großprojekte, deren scheitern Software unzuverlässig erscheinen lässt. Als Beispiel kann der NASA Mars Climate Orbiter angeführt werden. Er verglühte 1999 in der Atmosphäre des Planeten, nachdem metrische und imperiale Einheiten ohne Konvertierung zwischen Teilsystemen ausgetauscht wurden [3]. Dieses Problem hätte durch umfangreiche Tests entdeckt werden können. Unglücklicherweise lassen sich zahlreiche Projekte finden, die aufgrund fehlerhafter bzw. nicht ausreichend getesteter Software scheiterten [4].

Aber nicht gescheiterte Großprojekte, sondern alltägliches Versagen von Software stellt das wirkliche Problem dar [5]. Offenbar wächst die Komplexität von Softwaresystemen schneller, als Maßnahmen zur Beherrschung entwickelt werden [6]. Probleme finden sich in allen Arten von Entwicklungsprojekten. Allerdings

scheitern Projekte natürlich nicht generell; vielmehr entwickeln viele Unternehmen Software mit großem Erfolg. Wir regen daher an, aussichtsreiche Entwicklungsprojekte zu beobachten und erfolgreiche Vorgehensweisen abzuleiten.

In einem Projekt in Zusammenarbeit mit regionalen Unternehmen wollten wir erfahren, was Softwareentwicklung *erfolgreich* macht. Nach Herausarbeitung des Status quo haben wir dazu die gewonnenen Erkenntnisse ausgewertet und schließlich Handlungsempfehlungen abgeleitet. Wir haben für jede Empfehlungen erarbeitet, unter welchen Bedingungen und mit welchen Voraussetzungen sie umzusetzen ist. Denn die tatsächliche Umsetzung ist selbst für bekannte Vorgehensweisen nicht notwendigerweise einfach. Vielmehr ist Kontextwissen erforderlich, das sie für Unternehmen nutzbar macht. Die in diesem Artikel präsentierten Empfehlungen haben einen technischen Fokus oder beziehen sich auf bestimmte Technologien. Handlungsempfehlungen für die organisatorische Einbettung des Softwaretestens haben wir in [6] beschrieben.

Dieser Artikel ist wie folgt strukturiert. Abschnitt 2 führt in das Projekt ein. Die Forschungsmethodik wird in Abschnitt 3 skizziert. Unser Ordnungsrahmen zur Kategorisierung wird in Abschnitt 4 vorgestellt. In Abschnitt 5 diskutieren wir fünf technische Handlungsempfehlungen. Schließlich ziehen wir in Abschnitt 6 ein Fazit und weisen auf zukünftige Arbeiten hin.

2 Hintergrund

Münster und das umgebende Münsterland liegen in Nordrhein-Westfalen. Die Region weist eine hohe Zahl von Unternehmen mit Bezug zur Informationstechnologie auf. Die meisten dieser Firmen sind mittelständisch und entwickeln Software. Einige der größeren Firmen entwickeln zwar ebenfalls Software, dies erfolgt allerdings nur zur Unterstützung der eigenen Geschäftsprozesse. Es handelt sich dabei vor allem um Finanzdienstleister. Alle Unternehmen sind Mitglieder der Industrie- und Handelskammer, die das *Institut für Angewandte Informatik* (IAI) unterstützt. Das IAI ist der Westfälische Wilhelms-Universität angeschlossen und vereint die interdisziplinäre Arbeit von Ökonomen und Informatikern.

Der regelmäßige Austausch von Forschern des IAI und interessierten Unternehmen dient dazu, typischer Probleme der Wirtschaft aufzudecken. Auf dieser Weise erfuhr das IAI von einer generelle Unzufriedenheit mit dem Testen von Software. Die meisten Unternehmen versuchten, die Qualität der entwickelten Software zu erhöhen und gleichzeitig Kosten einzusparen. Nur bedingt war ihnen dabei bekannt, wie diese Ziele zu erreichen waren. Zudem fehlt Unternehmen in der Regel die Zeit, neue Technologien auszuprobieren oder Änderungen der Prozesse zu evaluieren. Nichts desto trotz war festzustellen, dass im Münsterland sehr erfolgreich Software entwickelt wird. Folglich zogen wir zwei Schlüsse [6]: Keines der Unternehmen hat einen *perfekten* Testprozess; alle stehen vor mehr oder weniger schweren Problemen. Jedes Unternehmen hat allerdings individuelle Stärken entwickelt, die dabei helfen, *gute* Software zu entwickeln.

Schließlich wurde das IAI-Projekt zur Verbesserung des Softwaretestens initiiert. Als Ziele wurden festgelegt, dass zunächst der Status quo des Testens in

der Region ermittelt werden sollte, um in der Folge erfolgreiche Vorgehensweisen ("best practices") zu identifizieren. Aus diesen sollten dann Handlungsempfehlungen abgeleitet werden. Es wurde erwartet, dass Stärken komplementär sind und dementsprechend zwar einige bereits bekannte Vorgehensweisen, aber auch viele interessante neue Ansätze gefunden werden konnten.

Die Heterogenität der teilnehmenden Unternehmen und der Softwareentwicklung im Allgemeinen ist sowohl Fluch als auch Segen. So lassen sich Vorgehensweise finden, die als Handlungsempfehlungen von hohem Wert für Unternehmen sind. Gleichzeitig erfordern die meisten Empfehlungen bestimmte Voraussetzungen, damit eine Umsetzung überhaupt möglich wird. Aus diesem Grund haben wir einen Ordnungsrahmen entwickelt, der Unternehmen helfen soll, die für sie relevanten Handlungsempfehlungen auszuwählen. Er wird in Abschnitt 4 erläutert.

Die beiden Ziele des Projekts, also die Ermittlung des Status quo und das Ableiten von Handlungsempfehlungen, lassen sich individuell darstellen. Daher bezieht sich dieser Artikel auf Handlungsempfehlungen, insbesondere auf solche, die technischen Aspekte des Testens oder verwendete Technologien betreffen.

3 Forschungsmethodik

Die für das Projekt gewählte Forschungsmethodik musste zwei Ziele miteinander verbinden. So sollte die Forschung akademischen Ansprüchen genügen, aber für Unternehmen direkt verwertbare Ergebnisse liefern. Aus diesen Gründen kommt ein gestaltungsorientierter Ansatz zum Einsatz, der in der englischsprachigen Literatur als *design science* bezeichnet wird. Das typischer Vorgehen zielt darauf ab, bisher ungelöst Probleme auf neue oder einzigartige Art und Weise anzugehen und effizienter oder effektiver zu lösen [7]. Dabei ist klar, dass es unmöglich ist, einen *idealen* Testprozess zu beschreiben oder eine ganzheitliche Beschreibung von Testmethoden zu liefern. Vielmehr sollen Empfehlungen erarbeitet werden, die möglichst viele typische Probleme adressieren und zu deren Lösung beitragen.

Erfolgreiche Vorgehensweisen werden sicherlich nicht gefunden, wenn Projektteilnehmern ein einfacher Fragebogen vorgelegt wird. Um die Probleme aus Sicht der Teilnehmer beleuchten zu können, haben wir semi-strukturierte Experteninterviews durchgeführt. Durch diesen qualitativen Ansatz, dem nur ein grober Interview-Leitfaden zugrunde lag, erfuhren wir, *wie* in den Unternehmen getestet wird. Im Verlauf der Interviews wurden dann Stärken und Schwächen der einzelnen Unternehmen thematisiert und schließlich erfolgreiche Vorgehensweisen mit den Teilnehmern erörtert.

Die von uns ermittelten Handlungsempfehlungen zielen nicht darauf ab, mit theoretischer oder praktischer Literatur zu Softwarequalität zu konkurrieren. Vielmehr sollen sie den vorhandenen Kenntnisstand ergänzen, der nicht für die Lösung aller in Unternehmen anzutreffenden Probleme ausreicht. In dieser Arbeit wird technischen Aspekten ein besonderer Fokus gegeben. Auch wenn organisatorische Fragen für erfolgreiches Testen gelöst werden müssen, sollte darauf geachtet werden, dass IT-Forschung versucht, neue Erkenntnisse in Bezug auf Informations*technologie* zu gewinnen [8].

Das grundlegende Vorgehen während des Projekts stellt sich wie folgt dar. Zunächst wurden Unternehmen, die das IAI unterstützen, kontaktiert und für Interviews geeignete Mitarbeiter identifiziert. Dabei wurden sowohl Führungskräfte als auch technisch ausgebildete Angestellte ausgewählt. Im zweiten Schritt folgten die Interviews. Mit kleinen Unternehmen wurde in der Regel nur ein Termin vereinbart. Große Unternehmen wurden hingegen mehrfach besucht, so dass Gespräche mit verschiedenen Mitarbeitern geführt werden konnten. In den Interviews sollte dabei geklärt werden, bei *wem* die Testverantwortung liegt, *wann* getestet wird, *was* dabei in die Test eingeschlossen ist (z. B. die graphische Benutzerschnittstelle oder die Geschäftslogik), *welche* Methoden genutzt wurden und *wie* das Testen im Allgemeinen angegangen wird. Des Weiteren haben wir versucht, möglichst viel über den Einsatz von *Testwerkzeugen* zu erfahren.

Nach Ermittlung des Status quo wurde mit den Teilnehmern diskutiert, auf welche generellen Probleme sie gestoßen sind und welche erfolgreichen Vorgehensweisen im Unternehmen entwickelt wurden. In diesem Zug wurde auch erörtert, welche Verbesserungen wünschenswert waren. Schließlich wurden potentiell ableitbare Handlungsempfehlungen besprochen. Dieser letzte Teil der Interviews wurde sehr offen gehalten, so dass zahlreiche Ideen ausgetauscht und viele interessante Ansätze besprochen werden konnten. In der Folge wurden die gewonnenen Erkenntnisse von uns verdichtet und anschließend analysiert. Darauf aufbauend konnte der Status quo gezeichnet und Handlungsempfehlungen zusammengestellt werden. Dabei achteten wir insbesondere darauf, die Bedingungen festzustellen, unter denen Empfehlungen einsetzbar sind.

4 Ordnungsrahmen

Um Handlungsempfehlungen für ein Thema, dass sehr komplex ist und zahlreiche Abhängigkeiten und Querverbindungen aufweist, nutzbar zu machen, bedarf es einer leistungsstarken Kategorisierung. Wirklich nützlich und für Unternehmen interessant werden Empfehlungen erst, wenn erklärt wird, *wie* sie zu nutzen sind und welche Voraussetzungen für die Nutzung erfüllt sein müssen. Aus diesem Grund wird der von uns entwickelte Ordnungsrahmen [6] verwendet.

Der *Anspruch* einer Empfehlung beschreibt, wie umfangreich organisatorische Veränderungen für die Umsetzung ausfallen. *Grundsätzliche* Empfehlungen sind so einfach umzusetzen, dass jedes Unternehmen sie befolgen sollte. Wenn der Aufwand darüber hinaus deutlich steigt, werden sie als *fortgeschritten* eingeordnet. Der *Zielzustand* beschreibt Ideale, die nur mit größtem Aufwand erreicht werden können. Da sie häufig die Initiierung eines Prozesses kontinuierlicher Verbesserungen erfordern, sollten Unternehmen abwägen, ob sie dieses Ziel anstreben wollen. Die erreichbaren Vorteile sind auf lange Sicht aber erheblich.

Ebenso wichtig ist die *Projektgröße*. *Kleine* Projekte werden meistens von einem einzelnen Team bearbeitet, das für Entwicklung und Testen zuständig ist. In *mittleren* Projekten werden diese Aufgaben in der Regel auf mindestens zwei Teams vergeteilt. *Große* Projekte weisen eine veränderte Struktur auf und vereinen häufig die Arbeitskraft hunderter Mitarbeiter. Auch könnten weitere Abtei-

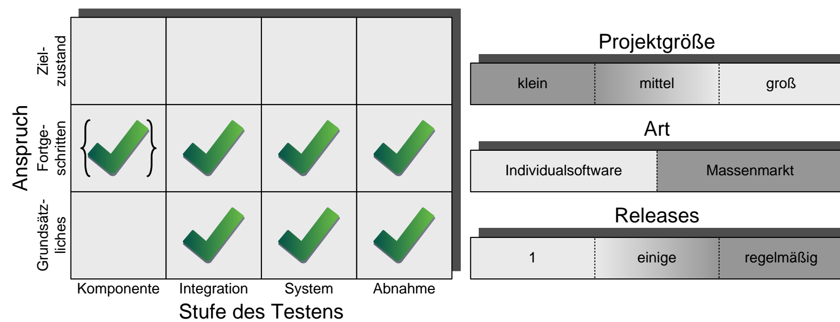


Abbildung 1. Beispielhafte Anwendung des Ordnungsrahmens

lungen beteiligt sein. Bei der Einordnung der Projektgröße sollte der Charakter des Projekts betrachtet werden, weniger die tatsächlichen Mitarbeiterzahlen.

Grob lassen sich zwei *Arten* von Softwareprodukten unterschieden. *Individualsoftware* wird gemäß Vertrag für einzelne Kunden implementiert, die oftmals in das Projekt eingebunden werden. Software für den *Massenmarkt* wird hingegen für einen größeren Kundenkreis entwickelt und über einen längeren Zeitraum angeboten. Folglich spielen hier Regressionstests eine große Rolle. Viele – aber nicht alle – Empfehlungen sind für beide Arten von Software nützlich.

Eine weitere Kategorisierung ist anhand der Zahl der *Releases* möglich. Es wird zwischen *einem*, *einigen* und *regelmäßigen* Releases unterschieden. Letzteres bedeutet, dass ein Produkt über Monate oder Jahre aktualisiert wird.

Die letzte Determinante teilt das Testen in *Stufen*. Der Literatur folgend [9] werden *Komponenten-*, *Integrations-*, *System-* und *Abnahmetest* unterschieden.

Wie in Abb. 1 dargestellt, bilden Anspruch und Stufe eine Matrix. Ein Häkchen zeigt dabei an, dass eine Handlungsempfehlung für die entsprechende Kombination Bedeutung hat. Wird es in Klammern dargestellt, ist die Bedeutung nebenrangig bzw. eine Implementierung für diese Kombination optional. Die anderen drei Determinanten sind als Balken dargestellt. Eine dunkelgraue Schraffur weist darauf hin, dass die Empfehlung unter dieser Bedingung gilt. Ein Farbverlauf deutet auf eine eingeschränkte Relevanz hin. In Abb. 1 ist die beispielhafte Verwendung des Ordnungsrahmens dargestellt:

- Die Handlungsempfehlung bezieht sich auf den Integrations-, System- und Abnahmetest. Sie hat sowohl einen grundsätzlichen als auch einen fortgeschrittenen Anspruch. Die Empfehlung sollte also umgesetzt werden, aber nicht notwendigerweise direkt im vollen Umfang.
- Das Häkchen beim Komponententest ist in Klammern dargestellt. Er profitiert zwar von einer Umsetzung, aber in geringerem Maße als andere Phasen.
- Die Handlungsempfehlung bezieht sich vor allem auf kleine und mittelgroße Projekte. Der Übergang der Färbung bedeutet, dass die Relevanz für solche Projekte noch gegeben ist, die kleinen Projekten ähnlich sind.
- Ferner bezieht sie sich ausschließlich auf Massenmarkt-Software. Theoretisch könnte sich hier für die individuelle Entwicklung ein Farbverlauf finden. Das hieße, sie würde ebenfalls profitieren, wenn auch in nicht so hohem Maße.

– Die entwickelte Software sollte zumindest in einigen Releases erscheinen, oder dauerhaft aktualisiert werden.

Für eine endgültige Umsetzungsentscheidung benötigen Unternehmen natürlich weitere Informationen. Der dargestellte Ordnungsrahmen kann aber helfen, einen Überblick zu bekommen und wichtige Voraussetzungen abzuschätzen.

5 Technische Empfehlungen

5.1 Moderne Entwicklungsumgebung

Die erste Handlungsempfehlung ist sicherlich wenig überraschend. Wir regen die Nutzung moderner Entwicklungsumgebungen an, insbesondere einer integrierten Entwicklungsumgebung (IDE) die anpassbar ist und sich über Plug-ins erweitern lässt. Die sich daraus ergebenden Vorteile sind sehr vielfältig. Die Nutzung bietet sich vor allem an, weil Software zur Entwicklung ohnehin genutzt wird und viele IDEs oder zumindest Plug-ins für sie *frei* erhältlich sind.

Der Einsatz einer IDE dient nicht nur dem Testen, kann aber signifikant zur Steigerung der Softwarequalität beitragen. Darüber hinaus unterstützt es Entwickler bei alltäglichen Aufgaben. Dies hat zur Konsequenz, dass sich weniger Programmierfehler durch Nachlässigkeiten einschleichen. Tester werden durch solche Fehler dann nicht aufgehalten, sondern können sich auf das Finden schwerwiegender Probleme konzentrieren. Diese Empfehlung dient folglich dem Testen, auch wenn sich nicht alle Vorteile *direkt* darauf beziehen.

Für Entwickler in kleinen Firmen ist es üblich, moderne IDEs zu nutzen. Unserer Erfahrung nach nutzen Unternehmen sie nicht notwendigerweise. Sobald die Wahl der Entwicklungssoftware nicht mehr in den Händen der Entwickler liegt, sondern es Richtlinien oder sogar Vorschriften gibt, kommt Software zum Einsatz, die hinter den aktuellen Möglichkeiten bleibt. Dies ist insbesondere dann zu beobachten, wenn PCs zentral installiert werden. Die Entwickler tauschen in einem solchen Fall nur selten die Programme aus und sind in vielen Fällen dazu noch nicht einmal in der Lage, etwa aufgrund Abhängigkeiten in der Systemlandschaft. Die von einigen Projektteilnehmern geschilderte Lage erinnerte uns an die 1980er Jahre. Entwickelt wurde ohne *Syntaxhervorhebung* (syntax highlighting) und die Entwicklungsumgebung bot kaum unterstützende Funktionalität. Die Dokumentation war nicht zugänglich. Vielmehr benutzen Entwickler direkt das Internet oder griffen auch für einfachste Fragen auf Bücher zurück. Am ungünstigsten stellte sich allerdings das Fehlern eines Debuggers dar. Debuggen erfolgte mithilfe von print()-Anweisungen, die zur mehr oder weniger zufälligen Ausgabe von Quelltextfragmenten führen.

Da wir sahen, wie viel produktiver *mit* einer modern IDE entwickelt werden und wie sehr dies die Softwarequalität erhöhen kann, empfehlen wir deren Nutzung dringend. Diese Empfehlung gilt für alle Unternehmen. Sie ist insbesondere für Komponententests hilfreich (vgl. Abb. 2). Diese Empfehlung gilt sogar dann, wenn die vorhandene Entwicklungsumgebung nicht erweitert oder durch Plug-ins ergänzt werden kann und ein Upgrade nötig wird. *Eclipse* etwa, die wohl

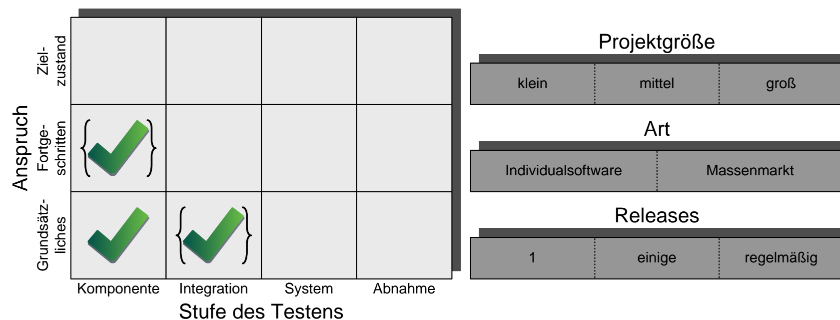


Abbildung 2. Einordnung der ersten Handlungsempfehlung

am weitesten verbreitete und gleichzeitig eine der leistungsstärksten IDEs, unterstützt Java, C/C++ und – über Erweiterungen – viele andere Sprachen. Zur umfangreichen Funktionalität kommt die Möglichkeit der Funktionsausbaus. Es steht eine vierstellige Zahl Plug-ins zur Verfügung (vgl. z. B. [10]).

Um die Leistungsfähigkeit der verwendeten Entwicklungsumgebung einschätzen zu können, empfiehlt sich ein Vergleich mit führenden IDEs. Die von einigen Projektteilnehmern eingesetzten Produkte blieben weit hinter dem zurück, was Eclipse oder Microsoft *Visual Studio* (um den Marktführer für die .NET-Entwicklung zu nennen) bereits vor Jahren leisteten.

Zur üblichen Funktionalität gehören die farbliche Hervorhebung des Quelltextes [11] und automatisierte Empfehlungen während der Eingabe (*code completion*). Des Weiteren lässt sich die Dokumentation direkt einblenden um etwa die Verwendung von veralteten Methoden (*deprecated*) zu verhindern. Auch lässt sich Code direkt auf Fehler überprüfen. Die *partielle Kompilierung* zeigt Hinweise, ohne den Compiler explizit aufzurufen. Software mit Syntaxfehlern kann so erst gar nicht kompiliert und vom Entwickler als *fertig* betrachtet werden.

Auch wenn die semantische Korrektheit unmöglich automatisch garantiert werden kann, lassen sich typische Fehler beispielsweise durch Warnungen vermeiden. Eclipse kann Java-Warnungen einblenden und betroffenen Code gelb unterschlängeln. Eine Variable etwa, die den Wert `null` annehmen kann aber ohne Prüfung verwendet wird, würde markiert werden. Somit kann Code, der `NullPointerExceptions` provoziert, korrigiert werden – ebenso wie viele weitere, potentielle Probleme. Entwickler gewöhnen sich daran schnell, selbst wenn Warnungen zunächst lästig erscheinen. Überflüssige Warnungen können zudem deaktiviert werden, z. B. durch Annotationen [12].

Als nächste Ausbaustufe bietet sich die Überprüfung so genannter *code rules* an. Kein IDE bietet diese Funktionalität direkt, sie kann aber über Plug-ins nachgerüstet werden. Während Warnungen vom Compiler stammen, verarbeiten diese Werkzeuge den Code selbst. Sie sind nicht nur sehr leistungsfähig, sondern auch anpassbar und somit für die Einführung von unternehmensweiten Programmierkonventionen geeignet. Solche Standard sind sehr empfehlenswert, auch wenn sich Entwickler nicht bevormundet fühlen sollten. Denn zahlreiche Probleme entstehen dadurch, dass mehrere Programmierer am selben Code arbeiten und ihn falsch interpretieren. Unternehmensweite Konventionen beugen

dem vor. Die entsprechenden Werkzeuge können zudem die Einhaltung der empfohlenen Programmierstandards der Hersteller (z. B. [13]) und der Literatur (z. B. [14]) sicherstellen.

In hohem Maße empfehlen wir auch die Nutzung der Debugging-Funktionen moderner IDEs. *Trace Debugger* können den Status einer Programms zu einem beliebigen Zeitpunkt visualisieren. Zeiger können verfolgt und Variablen unmittelbar verändert werden. Ferner ist die Ausführung Schritt-für-Schritt möglich. Darüber hinaus lassen sich Kontroll- und Datenflussgraphen anzeigen. Zusammen mit der Kenntnis moderner Debugging-Techniken [15] ist der Debugger einer modernen IDE ein sehr vielseitiges und leistungsfähiges Werkzeug.

Zusammenfassend raten wir dringend die Nutzung einer modernen IDE an, selbst wenn dies zur Bedingung hat, alte Bibliotheken, Vorgehensweisen, Paradigmen und sogar Programmiersprachen aufzugeben. Im Rahmen dieses Prozesses sollten bindende Vorgaben für die Formatierung von Quelltext sowie Benennungskonventionen festgelegt werden. Die optimale Nutzung einer IDE kann zudem gut durch ein Studium praxisbezogener Literatur wie [12,16,17,18] unterstützt werden. Hier existiert eine Vielzahl zum Teil sehr guter Titel.

5.2 Nutzung moderner Paradigmen und Frameworks

Softwaresysteme werden immer umfang- und funktionsreicher und damit komplexer. Als Möglichkeit zur Beherrschung bietet sich ein höherer Abstraktions- und Modularisierungsgrad an, der durch moderne Programmierparadigmen und Rahmenwerke (*Frameworks*) erreicht werden kann. Deren Einsatz wird allen Unternehmen empfohlen. Echte Vorteile ergeben sich nur für den Abnahmetests und für kleinste Projekte nicht, ansonsten ist durchgehend mit positiven Effekten zu rechnen (vgl. Abb. 3).

Klassisch wurde Software ohne Modularisierung entwickelt. Auch wenn dies heutzutage nicht mehr üblich ist, findet sich häufig eine starke und starre Verzahnung verschiedener Schichten. Sinnvoll hingegen wäre eine klare Trennung bei der eine Kommunikation nur über definierte Schnittstellen erfolgt. Diese kommt insbesondere Integrations- und Systemtests entgegen. Können sich Tester tatsächlich darauf verlassen, dass Module nur über ihre Schnittstellen angesprochen werden, reichen Tests mit Kenntnis des Modulcodes in der Komponententestphase. Schnittstellenlose Durchgriffe auf Daten hingegen erschweren das Testen wesentlich. Eine Abstraktion einzelner Funktionen ihm Rahmen der Modularisierung erleichtert auch die Rolleneinteilung: Spezialisierte Tester können gemäß ihrer fachlichen Stärken eingesetzt werden.

Um dienstorientierte Systeme zu testen, wie sie im Rahmen dienstorientierte Architekturen (*SOA*) zunehmend häufig anzutreffen sind, ist es nötig, Geschäfts- und Darstellungslogik zu trennen. Dienste sollten stets so spezifiziert werden, dass sie von beliebigen Aufrufern genutzt werden können. Idealerweise sollten sie darüber hinaus zustandslos sein oder vom Status abstrahieren. Für einen Daten liefernden Dienst sollte es unbedeutend sein, ob diese von derselben Anwendung für ihre Geschäftslogik, von der Darstellungslogik zwecks Ausgabe, oder von

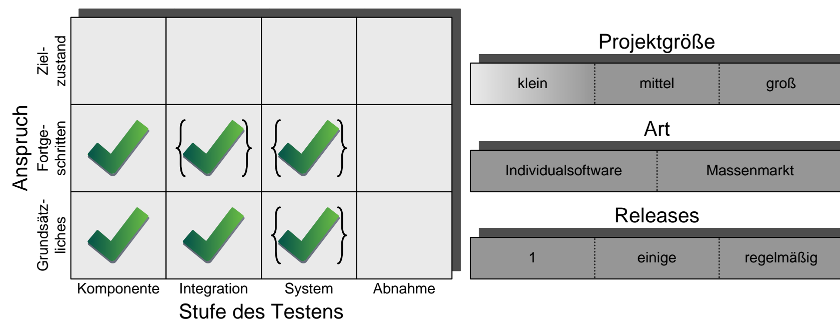


Abbildung 3. Einordnung der zweiten Handlungsempfehlung

einem Drittsystem abgefragt werden – sofern der Zugriff autorisiert ist. Nur dementsprechend definierte und nutzbare Dienste lassen sich effektiv Testen.

Eine fehlende Dienstorientierung ließ sich fast immer feststellen, wenn ältere Programmiersprachen zum Einsatz kamen, mit denen eine Kapselung schwierig ist. Aber auch bei modernen Programmiersprachen werden Frameworks benötigt. Denn ohne diese erfolgt häufig keine klare Schichtentrennung, selbst wenn die Sprache dies bieten würde. Die Nutzung moderner Programmiersprachen und -paradigmen wird dringend angeraten. Für Altsysteme (*legacy*) sollte geprüft werden, inwiefern eine Modularisierung und Dienstorientierung möglich ist. In der Regel ist es – zumindest über Erweiterungen – auch für Programmiersprachen wie COBOL oder Fortran möglich, Dienste anzubieten. In Frage käme auch ein *Wrapper*, der Dienste anbietet und das Altsystem maskiert.

Die große Zahl der angebotenen Literatur zur professionellen Programmierung verdeutlicht, dass auch erfahrene Programmierer die verwendete Sprache häufig weder effektiv noch effizient nutzen. Insbesondere scheint es weit verbreitet, alte Herangehensweise beizubehalten, selbst wenn diese durch *bessere* Möglichkeiten ersetzt wurden. Ein Auffrischen der Kenntnisse kann bei grundlegenden Techniken wie Entwurfsmustern [19,20,21] anfangen. Danach bietet sich Spezialliteratur zur effektiven Programmierung wie [12,16,22] an. Um gut testbaren Code zu schreiben ist es noch nicht einmal nötig, einen kompletten Wechsel auf einen testgetriebenen Entwicklungsprozess [23,24,25] zu vollziehen. Problematisch sind auch komplexe Programmiertechniken, die der Leistungssteigerung von Programmcode dienen sollen. Sicherlich ist es sinnvoll, diese im Rahmen eines Informatik-Studiums zu vermitteln. Auch steht außer Frage, dass es Bereiche gibt, in denen hardwarenah programmiert und in der jeder Kniff zur Leistungssteigerung realisiert werden sollte. Für allgemeine Projekte gilt dies nicht. Moderne Compiler sorgen für Optimierungen, die Bit-Shifts und ausgeklügelte arithmetische Operationen, die kaum lesbaren Code zur Folge haben, unnötig machen. Darüber hinaus werden geringe Effizienzgewinne schnell durch eine mangelnde Wart- und Testbarkeit wieder aufgehoben.

Zur Testbarkeit tragen auch moderne Frameworks bei. Ihre Stärke liegt insbesondere in der Abstraktion bzw. der vereinheitlichten Entwicklung. Darüber hinaus stellen sie häufig benötigte Funktionalitäten bereit. Deren Umsetzung ist bei verbreiteten Frameworks oftmals von hoher Qualität. Meistens ist eine

umfangreiche Dokumentation verfügbar. Laut Aussagen der Projektteilnehmer können Frameworks zur Entwicklung verteilter mehrschichtiger Applikationen wie *Enterprise Java Beans* (EJB) oder *Spring* das Testen wesentlich vereinfachen. Die Einarbeitung erfordert selbst bei leichtgewichtigen Frameworks ein Umdenken bei Entwicklern. Gerade dies kann aber dazu beitragen, unvorteilhafte Programmierpraktiken auf den Prüfstand zu stellen. Die Nutzung moderner Frameworks wird daher sehr empfohlen. Die Einführung sollte konsequent vollzogen und bei Notwendigkeit von Schulungen begleitet werden. Eine undisziplinierte, halbherzige oder gar falsche Verwendung der durch Frameworks zu Softwaresystemen ergänzten Komponenten kann zu massiven Problemen führen.

5.3 Enge Zusammenarbeit der Entwickler

Aufgrund der hohen Komplexität heutiger Softwareprojekte ist es nicht mehr zeitgemäß, Entwickler unabhängig voneinander Module implementieren zu lassen und diese dann in einer Integrationsphase zusammenzuführen. Vielmehr hat sich im Projekt gezeigt, dass eine enge Zusammenarbeit sehr hilfreich sein kann. Diese kann sich als Paarprogrammierung im Rahmen der agilen Entwicklung [26] darstellen, muss aber nicht unbedingt so weitgehend sein. Generell festhalten lässt sich, dass die niedrigere Effizienz durch eine höhere Personalbindung und mehr Austausch unter den Entwicklern durch eine höhere Code-Qualität und höhere Produktivität überkompensiert werden. Die Empfehlung, auf eine enge Zusammenarbeit hinzuwirken, gilt für alle Unternehmen. Vorteile zeigen sich vor allem für den Komponenten- und Integrationstests. Ein Start ist sehr einfach möglich, wobei sich umfangreiche Ausbaumöglichkeiten ergeben (vgl. Abb. 4).

Die enge Zusammenarbeit verfolgt eine Reihe von Zielen. Durch die gemeinsame Arbeit fallen Fehler schneller auf; besonders komplizierte Module können sogar in Paarprogrammierung implementiert werden. Die Einhaltung von Standards fällt einfacher (siehe auch Abschnitt 5.1). Wissen wird von Entwickler zu Entwickler weitergegeben und ergänzt die Dokumentation; idealerweise findet auch ein Erfahrungsaustausch statt.

Als besonders erfolgversprechend wurde uns von der gegenseitigen Begutachtung der Entwickler berichtet. Module werden nach wie vor in Einzelarbeit erstellt, vor Abschluss aber durch einen oder zwei weitere Mitarbeiter begutachtet. Es bietet sich dabei an, gefundene Fehler direkt zu beheben. Diese Arbeitsweise unter "Vier-Augen" ist nicht nur effektiv, sondern fördert auch den Austausch unter den Entwicklern. Selbst wenn Mitarbeiter ausfallen, findet sich für jedes Modul jemand, der es näher kennt. Ähnlich wie für die bisher vorgestellten Erfahrungen dient die gemeinsame Begutachtung nicht direkt dem Testen. Sie sorgt aber dafür, dass sich Tester auf das Finden schwerwiegender Fehler konzentrieren können, weil viele typische Probleme vor Beginn des Tests gelöst worden sind. Wichtig für die Begutachtung ist, sie als unterstützende und von Entwicklern begrüßte Maßnahme zu implementieren. Führen gefundene Fehler zu Sanktionen oder wird sie zur Leistungsmessung missbraucht, wird sie demotivierend wirken. Die Begutachtung von Quelltext wird auch als *Review* bezeichnet und ist seit

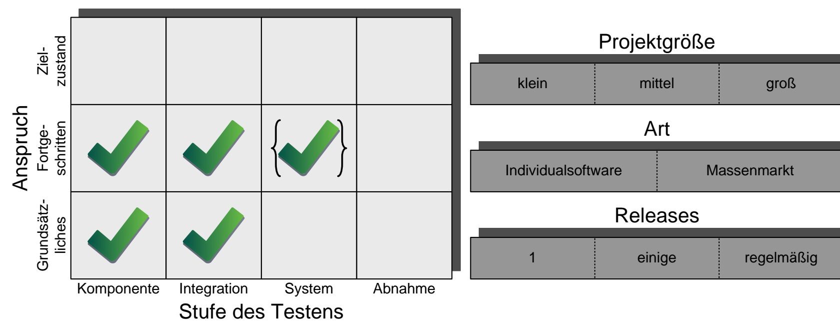


Abbildung 4. Einordnung der dritten Handlungsempfehlung

längerem bekannt [27]. Weitere Informationen finden sich im IEEE Standard für "software reviews" [28] sowie in praktischer Literatur wie [29,30,31].

Wichtig für die erfolgreiches Zusammenarbeit ist, dass die Verantwortlichkeiten klar geregelt sind. Die Zuordnung von Entwicklern und Begutachtern etwa muss mit Bedacht gewählt werden, denn es lassen sich sowohl Argumente für häufige Wechsel der Zuordnung wie für die Etablierung eingespielter Teams finden. Auch muss in diesem Zusammenhang festgelegt werden, welche Module überhaupt begutachtet werden.

5.4 Abstimmung von Testsystemen und Produktivsystemen

Die Nutzung moderner Programmiersprachen und Paradigmen für die Entwicklung komplexer verteilter Applikationen bietet nicht nur Vorteile. Werden Anwendungen auf Arbeitsplatzrechnern entwickelt aber auf Server oder Großrechner gespielt, kann es zu Kompatibilitäts- oder Skalierungsproblemen kommen.

Ursprünglich wurden Programme nur für ein Zielsystem entwickelt. Um auf anderen Plattformen zu laufen, war eine Anpassung nötig. Heutzutage sind Entwicklungs- und Zielplattform für gewöhnlich unterschiedlich. Auch läuft Software nicht mehr nativ auf der Hardware. Hypervisor, Applikationsserver und weitere Komponenten bilden zusätzliche Schichten, was weitreichende Vorteile hat. Als Folge sind Zielsysteme aber häufig leistungsfähiger und auch anders mit Software ausgestattet. In einfachen Fällen gilt dies nur für das Betriebssystem. Weitere Komponenten wie Bibliotheken, Datenbankmanagementsysteme (DBMS) oder auch Applikationsserver unterscheiden sich aber ebenfalls häufig.

Typische Kompatibilitätsprobleme lassen sich gut mit einem Beispiel motivieren. Java EE-Applikationen werden von Applikationsservern bereitgestellt. Auf Großrechnern und Servern kommen dabei Systeme wie IBM *WebSphere* zum Einsatz. Da sie auf "normalen" PCs erst gar nicht laufen, werden dort leichtgewichtige Alternativen wie Apache *Tomcat* verwendet. Auch wenn dies keine Probleme bereiten sollte, zeigt die praktische Erfahrung, dass offenbar durch unterschiedliche Interpretation von Spezifikationen, abweichende Versionen, konfliktäre Bibliotheken und ähnliches Inkompatibilitäten an der Tagesordnung sind.

Wir empfehlen daher die Abstimmung von Entwicklung- und Testsystemen mit der produktiven Umgebung. *Abstimmung* meint ein durchdachtes Vorgehen

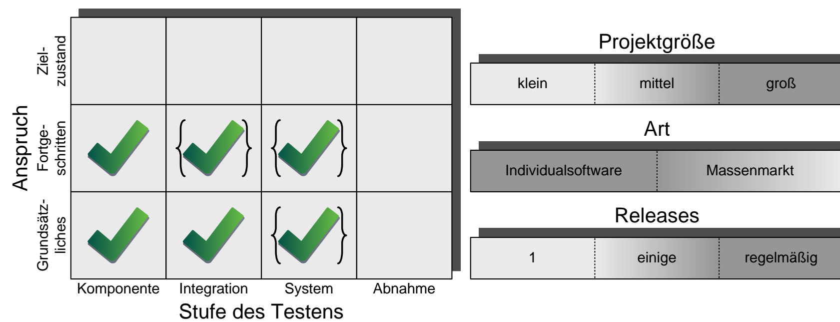


Abbildung 5. Einordnung der vierten Handlungsempfehlung

bei der Auswahl von Hard- und Software. Sie muss ökonomisch vertretbar bleiben – die Anschaffung eines Großrechners zum Testen wird in der Regel unmöglich sein, selbst wenn die Zielplattform ein solcher ist. In der Regel finden sich aber sinnvolle Lösungen. Die Abstimmung der Systeme wird ab mittelgroßen Projekten empfohlen, die in einige Releases münden. Vorteile zeigen sich vor allem in der individuellen Softwareentwicklung und den frühen Testphasen. (vgl. Abb. 5).

Aufgrund der Erfahrungen im Projekt halten wir auch höheren Aufwand für die Abstimmung für gerechtfertigt. Während sich Test- und Produktivsystem mit fortschreiten der Tests ohnehin annähern sollten, bietet es sich an, Kompatibilitätsprobleme so früh wie möglich zu lösen. Im obigen Beispiel kann eine "abgespeckte" WebSphere-Version für PCs genutzt werden. Tomcat als Testsystem sollte dann verwendet werden, wenn Tomcat auch auf dem Server läuft.

Anstatt ein DBMS auf dem Entwicklungssystem zu betreiben, kann fast immer das auf dem Server installierte genutzt werden. Zum Schutz der Daten wird eine neue Datenbank erstellt und vom restlichen Datenbestand entkoppelt. Heutige Server und vor allem Mainframes bieten zudem ausgeklügelte Virtualisierungsmöglichkeiten. Den produktiven Systemen identische Instanzen lassen sich einfach starten, um dann unter realistischen Bedingungen zu testen. Zu achten ist dabei allerdings auf die Ressourcennutzung, so dass Probleme während des Tests keine negativen Auswirkungen auf den Produktivbetrieb haben.

Auch umfangreiches Testen deckt häufig nicht alle Probleme auf. Dies gilt insbesondere für schlecht reproduzierbare Fehler, die in der Speichernutzung oder der parallelen Ausführung begründet liegen. Sie müssen sich auf Testsystemen nicht zeigen. Auch lässt eine akzeptable Leistung auf dem Testsystem keine Rückschlüsse auf die Leistungsfähigkeit auf der Zielplattform zu. Schließlich müssen komplexe Abhängigkeiten und wachsende Datenbestände ins Kalkül gezogen werden. Tests sollten aus diesen Gründen unbedingt auf der Zielplattform erfolgen. Defekte in parallelen Algorithmen – etwa drohende Wettlaufsituation (*race conditions*) – zeigen sich möglicherweise erst auf der leistungsfähigen Zielhardware oder aufgrund eines aggressiveren "Timings" der Zielsoftware.

Trotz aller Werbung für realistische Tests raten wir dringend, *nicht* ohne Vorbereitung auf produktiven Systemen zu testen. Produktive Daten dürfen nicht verändert werden. Die Gefährdung produktiver Daten oder ein "Ausbremsen" dieser Systeme würde alle Vorteile des realistischen Testens zunichte machen.

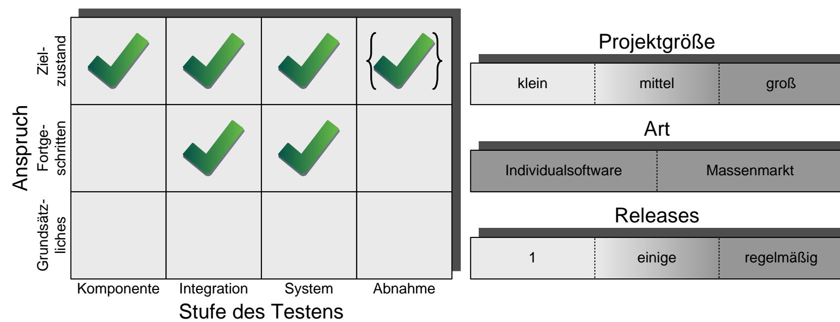


Abbildung 6. Einordnung der fünften Handlungsempfehlung

5.5 Integration der Werkzeuge

Der Einsatz von Testwerkzeugen ist mittlerweile üblich. Während die Nutzung von Unternehmen zu Unternehmen und in den unterschiedlichen Testphasen schwankt, konnten wir generell beobachten, dass Werkzeuge kaum integriert werden. Gerade das ist aber empfehlenswert.

Die meisten Testwerkzeuge sind eigenständige Applikationen. Nur einige integrierten sich als Plug-ins in IDEs. Allgemeine Formate oder Schnittstellen existieren kaum. Nur größere Anwendungen wie etwa die Produkte von *IBM Rational* bieten eine Austauschmöglichkeit von Daten. Diese bleibt meistens auf Produkte derselben Reihe beschränkt. Die Integration ihrer Werkzeuge wurde von vielen Projektteilnehmern angestrebt. Wie empfehlen sie ab mittelgroßen Projekten mit einigen Releases. Der Aufwand für die Integration is beachtlich. Auch wenn letztlich Vorteile für alle Phasen des Testens realisiert werden können, zeigen sich diese zunächst für Integrations- und Systemtests (vgl. Abb. 6).

Die Integration ist auf verschiedenen Ebenen erstrebenswert. Im ersten Schritt bietet es sich an, Systeme für die Dokumentation mit denen zur Testausführung zu verbinden. Undokumentierte Tests sind wenig wert und eine automatische Synchronisation der Ergebnisse mit der Dokumentation entlastet Tester wesentlich. Eine gut strukturierte Dokumentation [32] kann so einfacher erreicht werden. Die wachsende Datenbank kann anschließend für Auswertungen genutzt werden. Werte wie Laufzeiten oder Erfolgsraten sind für Testmanager sehr nützlich. Für regelmäßig aktualisierte Software kann dann als nächstes der *Bug Tracker* eingebunden werden, so dass gemeldete Fehler mit bekannten Defekten und vorhandenen Testfällen abgeglichen werden können.

Die Verbindung von Systemen zur Testausführung unterstützt Regressionstests. Testfälle aus früheren Stufen lassen sich einfach wiederholen. Die Anbindung des Testfallverwaltungsystems macht wiederholte Tests, die manuell nicht denkbar wären, wirtschaftlich. Natürlich ist eine solche Anbindung kompliziert und die Schritte bis zu einer automatisieren Regression mühsam. Nichts desto trotz halten wir die Regression für erstrebenswert. Durch ein Vorgehen in kleinen Schritten zur Verbesserung ist die Integration der Werkzeuge möglich, ohne das dies hohe Kosten oder Arbeitsaufwände verursacht.

Leider existieren keine Lösungen für eine umfangreiche Werkzeugintegration. Entsprechende Zusätze müssen selbst entwickelt werden, wobei Neuanschaffungen hinsichtlich ihrer Integrationsfähigkeit geprüft werden sollten. Schon kleine Werkzeuge, etwa für die Umwandlung von Daten, können viel manuelle Arbeit ersparen. Ein Beispiel ist ein Programm, das Ergebnisse aus der Testdokumentation extrahiert, aggregiert und Statistiken daraus erstellt. Es lässt sich schnell implementieren und kontinuierlich ausbauen. Dementsprechend sind vor allem Werkzeuge mit verfügbaren Quelltexten für die Integration geeignet. Sie lassen sich schnell den eigenen Wünschen anpassen und mit Schnittstellen versehen.

Die volle Integration bietet zahlreiche Vorteile. So könnte ein *Test Controlling* eingerichtet werden, das einen Überblick über den Testprozess verschafft und Kennzahlen berechnet [6]. Als Vision ergibt sich die Integration eines Systems, das Testfallverwaltung, Entwicklung- und Projektplanung, Testterminierung, Mitarbeiterzuweisung, Zeiterfassung, Aufgabenverwaltung und Controlling umfasst, sowie eventuell sogar ein *Management Cockpit*.

6 Fazit und zukünftige Arbeit

Wir haben in diesem Artikel die Ergebnisse eines Projektes vorgestellt, in dem wir zusammen mit regionalen Unternehmen Handlungsempfehlungen für die Softwareentwicklung und insbesondere das Testen erarbeitet haben. Dazu haben wir den Projekthintergrund beschrieben, einen Ordnungsrahmen zur Kategorisierung vorgestellt und fünf Empfehlungen mit technischem Fokus beschrieben.

Moderne IDEs unterstützen die Entwicklung wesentlich. Sie ermöglichen ein Testen, dass schwerwiegende Fehler sucht, anstatt sich mit Nachlässigkeiten im Code beschäftigen zu müssen. Die Nutzung moderner Paradigmen und Frameworks trägt dazu bei, strukturierter zu entwickeln und die Komplexität von Software zu beherrschen. Arbeiten Entwickler eng zusammen fördert dies den Wissensaustausch und dient ebenso dem Testen. Die Abstimmung von Test- und Produktivsystemen beugt Problemen durch Inkompatibilitäten vor. Eine Integration der Testwerkzeuge ist sehr aufwändig, kann aber die Leistung und Effizienz von Tests stark erhöhen.

Das diesem Artikel zugrunde liegende Projekt ist noch nicht beendet. Vielmehr sollen die Ergebnisse in Zukunft mit den Teilnehmern gemeinsam betrachtet und diskutiert werden. Idealerweise könnte eine quantitative Studie angeschlossen werden, die der Validation der Handlungsempfehlungen dient.

Literatur

1. Naur, P., Randell, B.: Software Engineering: Report of a conf. spon. by the NATO Science Committee, Garmisch, Germany. Scientific Affairs Division, NATO (1969)
2. Dijkstra, E.: The humble programmer. Comm. of the ACM **15** (1972) 859–866
3. NASA: Mars climate orbiter mishap investigation board phase I report (1999)
4. Kopec, D., Tamang, S.: Failures in complex systems: case studies, causes, and possible remedies. SIGCSE Bulletin **39**(2) (2007) 180–184

5. Charette, R.N.: Why software fails. IEEE Spectrum **42**(9) (2005) 42–49
6. Majchrzak, T.A.: Best practices for the organizational implementation of software testing. In: Proc. of the 43th Annual Hawaii International Conf. on System Sciences, Computer Society Press (2010) To appear.
7. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Quarterly **28**(1) (2004)
8. Orlikowski, W.J., Iacono, C.S.: Research commentary: Desperately seeking the "IT" in IT research–a call to theorizing the IT artifact. Info. Sys. Research **12**(2) (2001) 121–134
9. Watkins, J.: Testing IT: an off-the-shelf software testing process. Cambridge University Press, New York, NY, USA (2001)
10. o. A.: Eclipse plugin central. Online: http://www.eclipseplugincentral.com/.
11. Cowlishaw, M.F.: Lexx—a programmable structured editor. IBM Journal of Research and Development **31**(1) (1987) 73–80
12. Bloch, J.: Effective Java. 2nd edn. Prentice Hall, Upper Saddle River (2008)
13. Sun Microsystems, Inc.: Code Conventions for the Java Programming Language. Online: http://java.sun.com/docs/codeconv/.
14. Sutter, H., Alexandrescu, A.: C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series). Addison-Wesley (2004)
15. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, San Francisco (2009)
16. Meyers, S.: Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition). Addison-Wesley Professional (2005)
17. Conway, D.: Perl – Best Practices: Die deutsche Ausgabe. O'Reilly, Köln (2006)
18. Ford, S.: Effizienter Programmieren mit Visual Studio: 250 Tipps, um Ihre Produktivität zu verbessern. Microsoft Press, Unterschleißheim (2008)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addision-Wesley, München (1995)
20. Bishop, J.: C# 3.0 Entwurfsmuster. O'Reilly, Köln (2008)
21. Freeman, E., Freeman, E., Sierra, K., Bates, B.: Entwurfsmuster von Kopf bis Fuß. O'Reilly, Köln (2006)
22. Nagel, C., Evjen, B., Glynn, J., Skinner, M., Watson, K.: Professional C# 2008. Wrox Press Ltd., Birmingham (2008)
23. Beck, K.: Test-Driven Development by Example. Addison-Wesley (2002)
24. Newkirk, J.W., Vorontsov, A.A.: Test-Driven Development in Microsoft .Net. Microsoft Press, Redmond, WA, USA (2004)
25. Westphal, F.: Testgetriebene Entwicklung mit JUnit & FIT. Dpunkt Verlag, Heidelberg (2005)
26. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
27. Fagan, M.E.: Design and code inspections to reduce errors in program development. IBM Systems Journal **15**(3) (1976) 182–211
28. IEEE: IEEE Std 1028: IEEE standard for software reviews, New York (1998)
29. Wong, Y.K.: Modern Software Review. IRM Press (2006)
30. Wiegers, K.E.: Peer reviews in software. Addison-Wesley, Boston (2002)
31. Thayer, R.H.: Software Reviews, Inspections and Audits: A Standards-based Guide. IEEE Computer Society, Washington, DC, USA (2009)
32. IEEE: IEEE Std 829-2008: IEEE standard for software and system test documentation, New York (2008)

An Algebraic Framework for Modeling and Processing Hyperdocuments

Volker Mattick
Lehrstuhl Logik in der Informatik, Fakultät für Informatik, TU-Dortmund
volker.mattick@tu-dortmund.de

1 Introduction

The most prominent languages for describing hyperdocuments are based on the Extensible Markup Language (XML[1]). This family of languages is normed by committees, initiated by the W3C[2], and became the quasi-standard in the World Wide Web. These xml-based languages are more the result of long discussion processes, influenced by industry, government and praticioneers, then of a rigoros language design. Maybe over time this standards may change, but a radical new design of hyperdocument description languages is not in sight and would be very time-intensive and expensive. So we need techniques, dealing with this, from the perspective of language design not optimal standard, that enables us to understand these standards better and tools that help developers to produce correct documents, that fulfill the users' needs.
A lot of theoretical research in this area comes originally from database theory. An xml-based document there is seen as a file of a semi-structured database, which is described by the according schema. So mainly structural aspects play a role. When seen as a hyperdocument also formatting aspects are crucial.

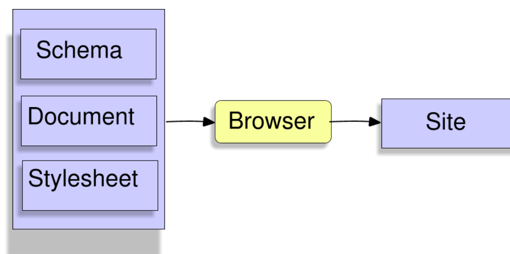


Fig. 1. *Simplified XML-Document Processing*

Abstracted from details (c.f. Fig. 1) a browser takes three different input files, a schema, a document and a stylesheet, each of them in a particular language. The browser parses the schema with an internally defined parser for the schema language and generates a parser for documents that are valid w.r.t. this schema. Then it reads the document itself and transfers it into an internal representation, called formatting object tree. In the next step the stylesheet is parsed with an internal predefined parser and transformed into functions, that modifies the formatting object tree into a refined formatting object tree. Finally this tree is interpreted by an area model, which is rendered by a layout engine (cf Fig. 2).
So in principle a browser is similar to an interpreter, parameterized with a grammar. An interpreter for a context-free grammar G can be defined by the according abstract syntax $\Sigma(G)$ and a parser for $L(G)$ and interpreted by a $\Sigma(G)$-algebra, which represents the target language. The common representation for both are the abstract syntax trees of G. Abstract syntax trees may not be the best form

[1] http://www.w3.org/XML
[2] World Wide Web Consortium, www.w3.org

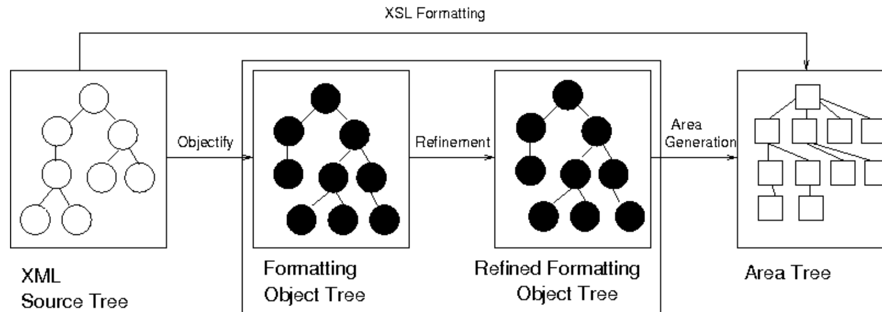


Fig. 2. *XSL Formatting*

to read for a human beeing, but it is the only unambigous form. So they can be used to represent and store xml-based hyperdocuments unambigously.

2 Preliminaries

A common method to define the *concrete syntax* of a formal language is via a grammar (c.f. [4]) that generates the language and which is often denoted in (extended) Backus-Naur-Form (BNF). Let now $L(G)$ be the language generated by the context-free grammar (CFG) G. Then the abstract syntax can be captured by a *many-sorted signature* $\Sigma = (S, F)$ (c.f. [2]), which consists of a set S of *sorts* and a $S^* \times S$-sorted set F of *function symbols*. Instead of $f \in F_{(e,s)}$, we write $f : e \rightarrow s \in F$. If $e = \epsilon$, then f is called a *constant*. In detail, if $G = (N, T, P, S)$ is a CFG, then the signature $\Sigma(G) = (N, CO)$ with $CO = \{c_p : X_1 \times \ldots \times X_n \longrightarrow X \mid \exists : p = (X \longrightarrow w_1 X_1 w_2 \ldots w_n X_n w_{n+1}) \in P, w_i \in T^*, 1 \leq i \leq n+1, X_j \in N, 1 \leq j \leq n\}$ is called the *abstract syntax of* G. A particular word of $L(G)$ is then represented by an *abstract syntax tree*. Let $\Sigma(G)$ be the abstract syntax of G, then $AST_{\Sigma(G)} = \cup_{n \in N} AST_{\Sigma(G)_n}$, where for all $e \in N^*, s \in N, f : e \longrightarrow s \in \Sigma(G)$ and $t \in AST_{\Sigma(G)_e}$ there holds $f(t) \in AST_{\Sigma(G)_s}$, is the set of all abstract syntax trees of G.

Each abstract syntax Σ can be assigned a semantic via a *Σ-structure*, a tuple (A, OP), where $\forall : s \in N$ there exists exactly one non-empty $A_s \in A$, $\forall : (co : s) \in CO$ there exists exactly one element $co^A \in A_s$ and $\forall : (co : e \rightarrow s) \in CO$ there exists exactly one function $co^A : A_e \rightarrow A_s \in OP$. Where possible without confusion, it is abbreviated as A. Mathematically, this structure is a Σ-algebra. One canonical algebra that exists for each Σ is the Σ-ground-term algebra, where each

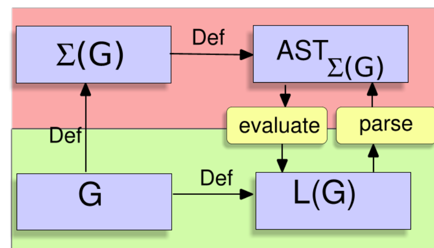


Fig. 3. *Concrete and abstract syntax*

constructor is interpreted symbolically. It can be shown that this is AST_Σ and it is initial in the class of all Σ-algebras. This means there exists an N-sorted Σ-homomorphism $eval^A : AST_\Sigma \rightarrow A$ for each Σ-algebra A. An s-sorted mapping h between Σ-structures A and B is called a *Σ-homomorphism*, if for all function symbols $f : w \longrightarrow s \in \Sigma$ there holds $h_s \circ f^A = f^B \circ h_w$. So each other semantic of the abstract syntax can be defined with a morphism from AST_Σ, called *evaluator*. If $AST_{\Sigma(G)}$ are the abstract syntax trees for a context-free grammar G, it can be shown that $L(G)$ is just one of these non-initial semantics of $\Sigma(G)$. Now for all $X \in N$, let $L(G)_X$ denote the subset of $L(G)$ that only contains sentences that are reachable from X, and $L(G) = \bigcup_{X \in N} L(G)_X$. So $L(G)$ is an N-sorted set that can be extended into a $\Sigma(G)$-algebra. To do that, define for each $c_p \in \Sigma(G)$, $c_p^{L(G)} : L(G)_{X_1} \times \ldots \times L(G)_{X_n} \longrightarrow L(G)_X$ as $c_p^{L(G)}(X_1[w_1/X_1], \ldots, X_n[w_n/X_n]) =_{def} X_1 \ldots X_n[w_1/X_1 \ldots w_n/X_n]$. Evaluator and parser are dual. A parser is a morphism between the two $\Sigma(G)$-structures$L(G)$ and $AST_{\Sigma(G)}$. But obviously there is no unique evaluator from $AST_{\Sigma(G)}$ to $L(G)$, whereas the parser is unambigous.
A more interesting feature is, that each semantic can be defined by a morphism and so a compiler in a target language can also be defined via a morphism or by defining an $\Sigma(G)$-structure and then showing that a morphism exists between $AST_{\Sigma(G)}$ and the new defined structure. So we have the possibility to give one abstract syntax tree different interpretations by only changing the $\Sigma(G)$-structure.

3 Algebraic Framework

An xml-based hyperdocument consists, beyond its name and some technical information, of three parts: the *schema*, the *stylesheet* and the structural *document* description. Each of this parts is in general described by a different language.

Schema: The schema S is the generating device for the language of the hyperdocument, similar to a CFG G is a generating device for a context-free language $L(G)$. Schemas generate only special cases of context-free languages, namely regular tree languages (c.f. [6], [5]), a subclass for which easily can be defined a balanced version (c.f. [1]). This balanced version is called *markup language* and denoted by $ML(S)$. Regular tree grammars conists only of guarded rules, which makes the abstraction quite easy, because we have a canonical name for each constructor. So in principle each markup language could be also defined by a CFG. The problem is, that a schema not only contains information, that can be captured by a CFG, e.g. in a schema can be defined how often an element can occur. For simple special cases that it occurs zero or one times, exact once or infinitly often we can find a context-free represenation, but characterizing a range in between is not possible, at least not with reasonable work. Also the fact, that an attribute is required and even more that it has a default value, can not captured by a CFG. So simply converting a schema into a CFG and then building a parser and interpreter for it will not work. In practice the schema is usually

described by a Document Type Definition (DTD), an XML-Schema (c.f. [9]) or a RelaxNG-pattern (c.f. [7]). All these languages can be described by CFGs and so they also have an abstract syntax. So, the solution is to construct the abstract syntax for the schema language and build a schema parser. The abstract syntax then can be interpreted by three different $\Sigma(S)$-algebras. The first one is the $ML(S)$-algebra, which interprets $AST_{\Sigma(S)}$ as the concrete markup language for the author of a document. The second one is the $\Sigma(ML(S))$-algebra, which interprets the schema as the abstract syntax for the markup language. The third one is the document parser algebra, which interprets $AST_{\Sigma(S)}$ as a parser for documents in markup language, including all the constraints that cannot expressed by the language itself. The parser must not only reject non well-formed documents, but also documents, which not fulfill the additional constraints given by the schema.

Stylesheet: The stylesheet is the formatter for a hyperdocument. It defines path-expressions to locate a particular place in the document tree and actions, which modify the selected part of the tree. A *document tree* is an unranked, sibling-ordered, labeled tree with has two different kinds of leaves, called *content nodes* and *attribute nodes*. This document trees are not the abstract syntax trees, but another interpretation of $AST_{\Sigma(ML(S))}$. In practice a stylesheet is usually described by CSS (c.f. [8]) or XSLT[3], both context free languages, which can be captured by an abstract syntax. CSS stylesheets can only modify values of attributes of the document tree, but not the document tree itself. XSLT is a tree-transformation language that can change also the structure of the tree.
For both languages a parser can be constructed that reads the concrete word of the stylesheet language $L(Style)$ into an abstract syntax tree of $AST_{L(Style)}$. So when a hyperdocument is represented by an element $doc \in AST_{\Sigma(ML(S))}$, then a stylesheet must be interpreted by paths on *doc* and the action by tree-transformations at the located place.

Document: The document is described by a word of the markup language $ML(S)$ and transformed via the parser algebra into an abstract syntax tree $AST_{\Sigma(ML(S))}$. The resulting abstract syntax tree is not necessarily complete in the sense, that all attributes occur and have values and not minimal in the sense that attributes can occur more then once. Moreover this AST is not formatted, because the stylesheet is not yet applied. In practice most hyperdocuments are described by (X)HTML or sublanguages of that.
The interpretation of a $doc \in AST_{\Sigma(ML(S))}$ must have default interpretations for missing attribute values, which may be influenced by information in the schema, e.g. given default values. If an attribute occur more then once the interpretation must always interpret only the last occurence.

Browser: The browser (c.f. Fig. 4) starts with parsing the schema $s \in S$, where S is the schema language, with an internal parser into an abstract syntax tree

[3] `http://www.w3.org/TR/xslt`

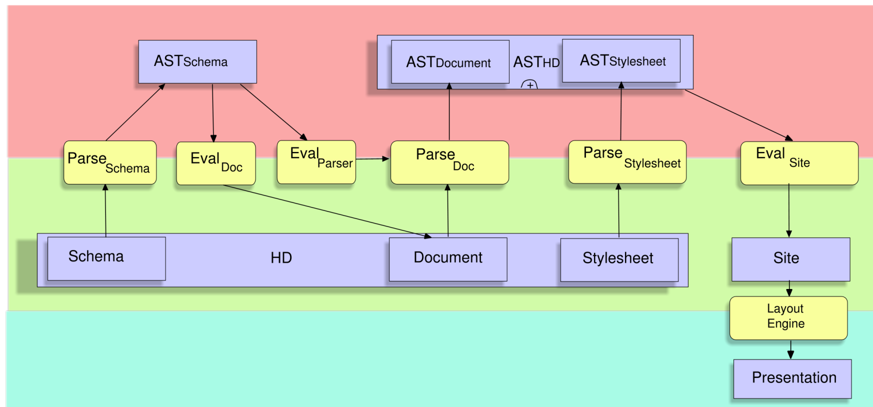


Fig. 4. *Complete Browser*

schema $\in AST_S$. In fact the abstract syntax trees here are not relevant, we are only interested in the signature $\Sigma(S)$ and in three special algebras. Firstly the markup language algebra, which interprets the signature as the according balanced regular tree language. Secondly the algebra, which interprets the signature as the abstract syntax. Thirdly the algebra, which interprets the signature as a validating parser for the markup language. The first one is needed for the author of the document, the second and third one is needed for the browser. All three algebras can be generated automatically, because they are constructed always by the same mechanisms. Then the document $d \in ML(S)$ itself is parsed by the parser algebra into *doc* $\in AST_{\Sigma(ML(S))}$. In this case we need the abstract syntax tree, because the parsed tree is not interpreted directly. Before we can add a semantic, we must parse the stylesheets. Here again the abstract syntax trees play no role, because we are only interested in one particular interpretation of the signature, namely that one, which interprets a stylesheet by a function on $AST_{\Sigma(ML(S))}$. Then the abstract syntax tree of the document is modified by the functions resulting from the stylesheet. This modified tree is then interpreted by a document algebra, we call *site* for short. For each document there can be different such algebras, depending on e.g. the later output media, the used layout engines or the reader's needs. Most browsers nowadays are graphical browsers, so the document itself is usually interpreted in a graphical way. But different browsers have slightly different interpretations, which leads to different output. For some special purposes, we need textual interpretation.

Editor: Usually hypertext editors work the way, that the author direct edits the markup language representation of the document. For pure textual editors this is not problematic, but for graphical editors it leads to vast problems, well known from WYSIWYG approaches. When a graphically designed document is directly stored in the markup language it is not necessarily interpreted the same way by another editor or browser (c.f. Fig. 5). For one graphical representation

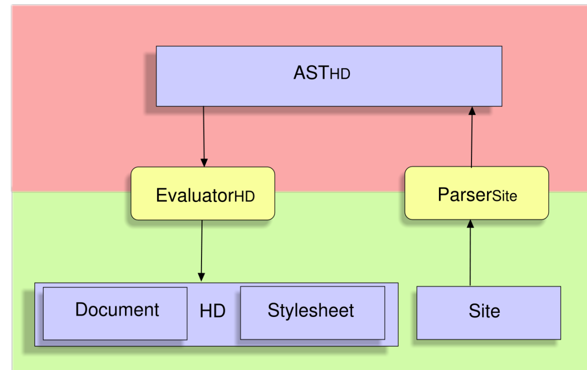


Fig. 5. *Editor*

it can exists more then one textual representation in a particular interpretation. In a even slightly different interpretation this may not be the same. If it is stored by the abstract syntax tree this problem is reduced. If the $\Sigma(ML(S))$-algebra is also stored this is unambigous.

Adaptable Hyperdocuments: Adaptable hyperdocuments (c.f. [3]) can be characterized by abstract syntax trees with variables. Usually abstract syntax trees are always ground terms, meaning terms without variables. If the interpretation of $AST_{\Sigma(ML(S))}$ is known, an abstract syntax tree with variables denotes a subset of $AST_{\Sigma(ML(S))}$. The second thing we have to know is a user profile, describing in terms of the $\Sigma(ML(S))$-algebra, which elements this algebra are allowed. Now the adaptable hypertext system must search for an assignment of the variables, so that the interpretion of the according variable-free abstraxt syntax tree fits the constraints of the algebra. It might be necessary that the original markup language $ML(S)$ must be enriched with new syntactical constructs for representing hyperdocuments with variabels, so called semi-documents.

4 Implementation

Here it is only given a short example using the GPS Exchange Format for exchanging geodata. This is not a typical markup language for hyperdocuments, but it has a comparatively short and understandable schema definition[4] and nearly all features can be demonstrated with this language too. The implementaion examples are coded with Haskell[5]
All elements and types of the schema are represented in the abstract syntax as constructors. The attributes, which belongs to a complex type are represented as constructors of the according attribute sort. Constraints like `minOccurs` or

[4] `http://www.topografix.com/gpx/1/1/gpx.xsd`
[5] `http://www.haskell.org`

`maxOccurs` have default value 1, so it means e.g. that `metadata` should be appear either exact one time or not at all. Basic types like `xsd:string` are assumed to have a predifined interpretation.
This following complex type from the schema S,

```
<xsd:element name="gpx" type="gpxType"/>}
<xsd:complexType name="gpxType">
<xsd:sequence>
<xsd:element name="metadata" type="metadataType" minOccurs="0"/>
<xsd:element name="wpt" type="wptType"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="rte" type="rteType"
             minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="trk" type="trkType"
             minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="version" type="xsd:string"
               use="required" fixed="MiniGPX"/>
<xsd:attribute name="creator" type="xsd:string" use="required"/>
</xsd:complexType>
```

then can be represented by a signature $\Sigma(S) = (N, OP)$:

$OP = \{$ root_ :: `gpxType_av` $\times$ `gpxType` $\rightarrow$ `gpx`
gpxType_ :: `metadataAlt` $\times$ `wptList` $\times$ `rteList` $\times$ `trkList` $\rightarrow$ `gpxType`
gpx_MiniGPX :: $\rightarrow$ `gpxType_av`
gpx_creator :: `xsd:string` $\rightarrow$ `gpxType_av`
_ : _ :: `gpxType_av` $\times$ `gpxType_avList` $\rightarrow$ `gpxType_avList`
[] :: $\rightarrow$ `gpxType_avList`
...

This can be implemented straightforward in Haskell the following way:

```
data GpxSIG gpx gpxType gpxType_av ... =
     GpxSIG {root_mt :: [gpxType_av] -> gpx,
             root_ :: [gpxType_av] -> gpxType -> gpx,
             gpxType_ :: (Maybe metadata) -> [wpt] -> [rte] -> [trk] -> gpxType,
             gpx_MiniGPX :: gpxType_av,
             gpx_creator :: XSD_string -> gpxType_av, ...}
```

The algebra, which interprets a document as an abstract syntax tree of $AST_{\Sigma(ML(S))}$ looks like this in the Haskell implementation:

```
gpxALG :: GpxSIG Gpx GpxType GpxType_av ...
gpxALG = GpxSIG Root_Mt Root_ GpxType_ Gpx_MiniGPX Gpx_creator ...
```

```
data Gpx = Root_Mt [GpxType_av] | Root_ [GpxType_av] GpxType
data GpxType = GpxType_' (Maybe Metadata') [Wpt] [Rte] [Trk]
data GpxType_av = Gpx_MiniGPX | Gpx_creator XSD_string
```

And a second algebra, which interprets a document as a object of the Scalable Vector Graphics (SVG[6]) can look like this:

```
svgALG = GpxSIG gpx_ gpxType_ gpxType_av_ metadata_ metadataType_ wpt_ wptType_
                rte_ rteType_ trk_ trkType_ ... =
         where root_mt _ = "<svg />"
               root_ avlist gpxType = "<svg xmlns=\"http://www.w3.org/2000/svg\"
                    xmlns:svg=\"http://www.w3.org/2000/svg\"
                    xmlns:xlink=\"http://www.w3.org/1999/xlink\">"
                    ++ gpxType ++ "</svg>"
               gpxType_ metadata wpt rte trk = "<g>" ++ (conc wpt)
                       ++ (conc rte) ++ (conc trk) ++ "</g>"
               gpx_MiniGPX = "MiniGPX"
               gpx_creator c = c
```

Then a monadic parser not only parses the document, but also transforms it directly into a given interpretation.

```
parseGpx :: GpxSIG gpx gpxType gpxType_av metadata metadataType wpt
    wptType wptType_av rte rteType trk trkType name time
    bounds boundsType_av ele sym number rtept trkseg trksegType trkpt
    -> MParser Char gpx
parseGpx alg = do result <- (parseE `parM` parseC); return result
        where parseE = do avlist <- (opencloseAV "gpx" (parseGpxType_av alg));
                          return (root_mt alg avlist)
              parseC = do avlist <- (openAV "gpx" (parseGpxType_av alg))
                          content <- (parseGpxType alg)
                          close "gpx"
                          return (root_ alg avlist content)

parseGpxType ::  ... -> MParser Char gpxType
parseGpxType alg = do result1 <- qmM' (parseMetadata alg)
                      result2 <- starM (parseWpt alg)
                      result3 <- starM (parseRte alg)
                      result4 <- starM (parseTrk alg)
                      return (gpxType_ alg result1 result2 result3 result4)

parseGpxType_av :: ... -> MParser Char gpxType_av
```

[6] http://www.w3.org/Graphics/SVG/

```
parseGpxType_av alg = parL [parseGpx_MiniGPX alg, parseGpx_creator alg]
     where parseGpx_MiniGPX alg = do isTag "version"
                                     isChar '='
                                     result <- parseDQ
                                     return (gpx_MiniGPX alg)
           parseGpx_creator alg = do isTag "creator"
                                     isChar '='
                                     result <- parseDQ
                                     return (gpx_creator alg result)
```

If parseGPX is called with gpxALG and applied on a document from $L(ML(S))$,

```
<gpx version="MiniGPX" creator="JOSM GPX export">
  <metadata>
    <bounds minlat="51.4813624" minlon="7.385542"
            maxlat="51.5019492" maxlon="7.4255655"/>
  </metadata>
  <trk>
    <name>H-Bahn</name>
    <trkseg>
      <trkpt lat="51.4921644" lon="7.4166625">
        <time>2008-03-28T17:02:06Z</time>
      </trkpt>
      <trkpt lat="51.4922462" lon="7.4167966">
        <time>2008-03-31T22:29:55Z</time>
      </trkpt>
      <trkpt lat="51.492271" lon="7.4168691">
        <time>2008-11-27T12:47:33Z</time>
      </trkpt>
    </trkseg>
  </trk>
</gpx>
```

then the result is an abstract syntax tree from $AST_{\Sigma(ML(S))}$.

```
Root_ [Gpx_MiniGPX,Gpx_creator "JOSM GPX export"]
  (GpxType_
    (Just (Metadata_ (MetadataType_
       Nothing
       Nothing
       (Just (Bounds_mt [Bounds_minlat 51.48136,Bounds_minlon 7.385542,
                         Bounds_maxlat 51.50195,Bounds_maxlon 7.4255657])))))
    []
    []
    [Trk_ (TrkType_ (Just (Name_ "H-Bahn"))
      [Trkseg_ (TrksegType_ [
       Trkpt_ [WptType_lat 51.492165,WptType_lon 7.4166627]
```

```
        (WptType_ Nothing (Just (Time_ "2008-03-28T17:02:06Z")) Nothing Nothing),
      Trkpt_ [WptType_lat 51.492245,WptType_lon 7.4167967]
        (WptType_ Nothing (Just (Time_ "2008-03-31T22:29:55Z")) Nothing Nothing),
      Trkpt_ [WptType_lat 51.49227,WptType_lon 7.416869]
        (WptType_ Nothing (Just (Time_ "2008-11-27T12:47:33Z")) Nothing Nothing)])])]))
```

If it is called with `svgALG`, the the result is a SVG-term:

```
<svg>
 <g>
  <line fill="none" stroke="#000000" stroke-width="2" x1="51.492165"
   x2="51.492245" y1="7.4166627" y2="7.4167967"/>
  <line fill="none" stroke="#000000" stroke-width="2" x1="51.492245"
   x2="51.49227" y1="7.4167967" y2="7.416869"/>
 </g>
</svg>
```

5 Conclusion

The presented algebraic approach gives a precise and clear model to understand xml-based hyperdocuments and hyperdocument processing. It shows new techniques for developing and implementing browsers and editors, as both can be seen as constructing compilers. Known techniques and results from this area can be adapted. The abstract syntax trees can be used to store xml-based hyperdocuments unambiguously, a great advantage to document trees. For the core topics a Haskell implementation shows the usability of the model.

References

1. Jean Berstel and Luc Boasson. XML Grammars. In *Mathematical Foundations of Computer Science*, pages 182–191, 2000. `http://www-igm.univ-mlv.fr/~berstel/Articles/XMLgrammars.ps`.
2. Manfred Broy, Martin Wirsing, and Peter Pepper. On the Algebraic Definition of Programming Languages. *ACM Trans. Program. Lang. Syst.*, 9(1):54–99, 1987. `http://doi.acm.org/10.1145/9758.10501`.
3. Paul de Bra. Design Issues in Adaptive Web-Site Development. In *Prroceedings of the 2nd Workshop on Adaptive Systems and User Modeling on the WWW*, volume 99-07 of *TUE Computing Science Report*, pages 29–39, 1999. `wwwis.win.tue.nl/~debra/asum99/debra/debra.html`.
4. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation. 3 Edition.* Addison-Wesley, 2007.
5. Wim Martens. *Static Analysis of XML Transformation and Schema Languages.* University of Antwerp, 2006. Ph.D. Thesis, `http://lrb.cs.uni-dortmund.de/~martens/data/phdthesis.pdf`.
6. Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Interet Technol.*, 5(4):660–704, 2005.

7. RELAX NG specification. Technical report, Organization for the Advancement of Structured Information Standards [OASIS], December 2001. http://relaxng.org/spec-20011203.html.
8. Cascading Style Sheets, level 2 revision 1. CSS 2.1 Specification. Technical report, World Wide Web Consortium, October 2006. W3C Working Draft 06 November 2006, www.w3.org/TR/CSS21.
9. XML Schema Definition Language (XSDL) 1.1 Part 1: Structures. Technical report, World Wide Web Consortium, August 2007. W3C Recommendation, http://www.w3.org/TR/2007/WD-xmlschema11-1-20070830.

Maintaining XML Data Integrity in Programs – An Abstract Datatype Approach

Patrick Michel
Technische Universität Kaiserslautern

XML technology can be supported in high level programming languages in many different ways. APIs like DOM and SAX offer support for data in the generic XML format. Data binding approaches like JAXB and language extensions like XJ offer support for the combination of XML with XML schemata. Utilizing the type system of languages to guarantee validity of XML data is one way to improve language support.

There is almost no support, however, for the combination of XML with more powerful schema languages like Schematron. Such languages allow the definition of arbitrary boolean expressions on the shape and more important the content of XML data. Tools supporting Schematron are able to check all these integrity constraints for any given document, but offer no support for the correct manipulation of such documents. Schematron also does not define any method of manipulation, but allows to define valid sets of documents only.

We propose to view XML documents with such complex constraints as abstract datatype, with an interface of schema-specific procedures. Such an interface integrates well with object-oriented languages like Java and encapsulates the alien aspects of tree-shaped data. By automatically guarding these procedures with minimal preconditions, the lasting correctness and validity of an XML document can be guaranteed.

The procedures are implemented in a language accessible to Java programmers and domain experts, incorporating abstractions of the XML domain. It is powerful enough to define basic atomic manipulations that can leave valid documents in a valid state. An automated analysis is able to derive minimal preconditions that check if a procedure is applicable to a valid document. Failure of these preconditions can be dealt with by the Java programmer, for example, by changing parameters or calling other procedures.

Together with typical data binding approaches, which allow the type correct and convenient navigation and reading on documents, our approach offers comprehensive support for XML data with complex integrity constraints within languages like Java.

Lambdas und Schleifen in monotonen Logikprogrammen

Ulrich Neumerkel
Technische Universität Wien

Lambda-Abstraktionen und Schleifenkonstrukte sind in der Logikprogrammierung bisher kaum aufgenommen worden. Diese Zurückhaltung hängt unmittelbar mit den algebraischen Eigenschaften der verwendeten Konstrukte zusammen. Die bisher vorgeschlagenen Ansätze verletzen grundlegende Eigenschaften wie die der Monotonie, wodurch nicht nur eine deklarative Betrachtungsweise verunmöglicht wird und diagnostische Verfahren stark eingeschränkt werden, sondern auch effiziente Implementierungen behindert werden.

Mit der vorgestellten Umsetzung von Lambda-Ausdrücken werden die meisten Mängel unmittelbar vermieden. Es eröffnen sich nun neue Möglichkeiten monotone Schleifenkonstrukte einzubinden.

Functional Program Verification in Theorema. Soundness and Completeness

Nikolaj Popov, Tudor Jebelean*

Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
{popov,jebelean}@risc.uni-linz.ac.at

Abstract. We present a method for verifying recursive functional programs. We define a Verification Condition Generator (VCG) which covers the most frequent types of recursive programs. These programs may operate on arbitrary domains. Soundness and Completeness of the VCG are proven on the meta level, and this provides a warranty that any system based on our results will be sound.

1 Introduction

We present an experimental prototype environment for defining and verifying recursive functional programs, which is part of the *Theorema* system. In contrast to classical books on program verification [6],[4],[10] which expose methods for verifying correct programs, we put special emphasize on verifying incorrect programs. The user may easily interact with the system in order to correct the program definition or the specification.

There are various tools for proving program correctness automatically or semiautomatically, (see, e.g., [12],[1],[2]), and this is where our contribution falls into. As a distinctive feature of our prototype is the hint on "what is wrong" in case of a verification failure.

This work is performed in the frame of the *Theorema* system [3], a mathematical computer assistant which aims at supporting all phases of mathematical activity: construction and exploration of mathematical theories, definition of algorithms for problem solving, as well as experimentation and rigorous verification of them. *Theorema* provides both functional as well as imperative programming constructs. Moreover, the logical verification conditions which are produced by the methods presented here can be passed to the automatic provers of the system in order to be checked for validity. The system includes a collection of general as well as specific provers for various interesting domains (e. g. integers, sets, reals, tuples, etc.). More details about *Theorema* could be found at `www.theorema.org`.

* This research was partly supported by BMBWK (Austrian Ministry of Education, Science, and Culture) and Upper Austrian Government Project "Technologietransferaktivitäten".

2 Programming, Specification and Verification

As usual, correctness is transformed into a set of first-order predicate logic formulae – verification conditions. As a distinctive feature of our method, these formulae are not only sufficient, but also necessary for the correctness [7]. We demonstrate our method on a relatively simple example, however, it show how correctness may be proven fully automatically. In fact, even if a small part of the specification is missing – in the literature this is often a case – the correctness cannot be proven. Furthermore, a relevant counterexample may be constructed automatically.

We consider the total correctness problem expressed as follows: *given* the program which computes the function F in a domain D and given its specification by a precondition on the input $I_F[x]$ and a postcondition on the input and the output $O_F[x, y]$, *generate* the verification conditions VC_1, ... , VC_n which are sufficient for the program to satisfy the specification. The function F satisfies the specification, if: for any input x satisfying I_F, F terminates on x, (we write $F[x] \downarrow$) and the condition $O_F[x, F[x]]$ holds:

$$(\forall x : I_F[x]) \ (F[x] \downarrow \ \wedge O_F[x, F[x]]). \tag{1}$$

A Verification Condition Generator (VCG) is a device—normally implemented by a program—which takes a program, actually its source code, and the specification, and produces verification conditions. These verification conditions do not contain any part of the program text, and are expressed in a different language, namely they are logical formulae.

Any VCG should come together with its *Soundness* statement, that is: for a given program F, defined on a domain D, with a specification I_F and O_F if the verification conditions VC_1, ... , VC_n hold in the theory $Th[D]$ of the domain D, then the program F satisfies its specification I_F and O_F.

Moreover, we are also interested in the following question: What if some of the verification conditions do not hold? May we conclude that the program is not correct? In fact, the program may still be correct. However, if the VCG is complete, then one can be sure that the program is not correct. A VCG is complete, if whenever the program satisfies its specification, the produced verification conditions hold.

The notion of *Completeness* of a VCG is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on “what is wrong”. Indeed, most books about program verification present methods for verifying correct programs. However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

3 Coherence and Verification Conditions

Before performing the "real" verification, we first make sure that our programs are coherent. It is not that programs which are not coherent are necessarily not correct, however, in order to construct a system of programs preserving modularity, we need to use only coherent programs.

3.1 Coherent Programs

In this subsection we state the principles we use for writing coherent programs with the aim of building up a non-contradictory system of verified programs. Although, these principles are not our invention (similar ideas appear in [8]), we state them here because we want to emphasize on and later formalize them.

Building up correct programs: Firstly, we want to ensure that our system of coherent programs would contain only correct (verified) programs. This we achieve, by:

– start from basic (trustful) functions e.g. addition, multiplication, etc.;

– define each new function in terms of already known (defined previously) functions by giving its source text, the specification (input and output predicates) and prove their total correctness with respect to the specification.

This simple inductively defined principle would guarantee that no wrong program may enter our system. The next we want to ensure is the easy exchange (mobility) of our program implementations. This principle is usually referred as:

Modularity: Once we define the new function and prove its correctness, we "forbid" using any knowledge concerning the concrete function definition. The only knowledge we may use is the specification. This gives the possibility of easy replacement of existing functions. For example we have a powering function P, with the following program definition (implementation):

$$P[x, n] = \textbf{ If } n = 0 \textbf{ then } 1 \textbf{ else } P[x, n-1] * x$$

The specification of P is:
The domain $\mathbb{D} = \mathbb{R}^2$, precondition $I_P[x, n] \iff n \in \mathbb{N}$ and a postcondition $O_P[x, n, P[x, n]] \iff P[x, n] = x^n$.

Additionally, we have proven the correctness of P. Later, after using the powering function P for defining other functions, we decide to replace its definition (implementation) by another one, however, keeping the same specification. In this situation, the only thing we should do (besides preserving the name) is to prove that the new definition (implementation) of P meets the old specification.

Furthermore, we need to ensure that when defining a new program, all the calls made to the existing (already defined) programs obey the input restrictions of that programs – we call this:

Appropriate values for the auxiliary functions. The following example will give an intuition on what we are doing. Let the program for computing F be:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x],$$

with the specification of F (I_F and O_F) and specifications of the auxiliary functions H (I_H and O_H), G (I_G and O_G). The two verification conditions, ensuring that the calls to the auxiliary functions have appropriate values are:

$$(\forall x : I_F[x])\ (Q[x] \implies I_H[x])$$
$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_G[x]).$$

3.2 Recursive Programs and Generation of Verification Conditions

As is well-known, there is no universal VCG. Thus, in our research, we concentrate on constructing a VCG which is appropriate only for a certain kind of recursive programs – those which are defined by multiple choice *if-then-else* with zero, one, or more recursive calls on each branch (but without nested recursion). They are defined as those F:

$$F[x] = \textbf{ If } Q_0[x] \textbf{ then } S[x] \tag{2}$$

$$\begin{aligned} &\textbf{elseif } Q_1[x] \textbf{ then } C_1[x, F[R_1[x]]] \\ &\textbf{elseif } Q_2[x] \textbf{ then } C_2[x, F[R_2[x]]] \\ &\ldots \\ &\textbf{else } Q_n[x] \textbf{ then } C_n[x, F[R_n[x]]]. \end{aligned}$$

where Q_i are predicates and S, C_i, R_i are auxiliary functions ($S[x]$ is a "simple" function (the bottom of the recursion), $C_i[x, y]$ are "combinator" functions, and $R_i[x]$ are "reduction" functions). We assume that the functions S, C_i, and R_i satisfy their specifications given by $I_S[x]$, $O_S[x, y]$, $I_{C_i}[x, y]$, $O_{C_i}[x, y, z]$, $I_{R_i}[x]$, $O_{R_i}[x, y]$. Additionally, assume that the Q_i predicates are non-contradictory, that is $Q_{i+1} \Rightarrow \neg Q_i$ and $Q_n = \neg Q_0 \wedge \cdots \wedge \neg Q_{n-1}$, which we do only in order to simplify the presentation.

Note that functions with multiple arguments also fall into this scheme, because the arguments x, y, z could be vectors (tuples).

Type (or domain) information does not appear explicitly in this formulation, however it may be included in the input conditions.

Considering Coherent Recursive programs, we give here the appropriate definition:

Let S, C_i, and R_i be functions which satisfy their specifications. Then the program (2) is coherent if the following conditions hold:

$$(\forall x : I_F[x])\ (Q_0[x] \implies I_S[x]) \tag{3}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_F[R_1[x]]) \tag{4}$$

$$\ldots$$

$$(\forall x : I_F[x])\ (Q_n[x] \implies I_F[R_n[x]]) \tag{5}$$

$$(\forall x : I_F[x])\ (Q_1[x] \implies I_{R_1}[x]) \tag{6}$$

$$\ldots$$

$$(\forall x : I_F[x])\ (Q_n[x] \implies I_{R_n}[x]) \tag{7}$$

$$(\forall x : I_F[x])(Q_1[x] \wedge O_F[R_1[x], F[R_1[x]]] \implies I_{C_1}[x, F[R_1[x]]]) \tag{8}$$

$$\ldots$$

$$(\forall x : I_F[x])(Q_n[x] \wedge O_F[R_n[x], F[R_n[x]]] \implies I_{C_n}[x, F[R_n[x]]]). \tag{9}$$

It is not that a program which is not coherent is necessarily not correct. However, non-coherent programs are somehow inconsistent, namely proving their correctness would involve knowledge about their auxiliary functions which is out of the official scope. Thus, if we allow them in our system of verified programs, the modularity would be lost.

After performing the coherence check, we go to the verification. The upcoming theorem gives the necessary and sufficient conditions for a program to be correct. These conditions are taken as the *Verification Conditions*.

Theorem 1. *Let S, C_i, and R_i be functions which satisfy their specifications. Let also the program (2) be coherent. Then, (2) satisfies the specification given by I_F and O_F if and only if the following verification conditions hold:*

$$(\forall x : I_F[x])\ (Q_0[x] \implies O_F[x, S[x]]) \tag{10}$$

$$(\forall x : I_F[x])(Q_1[x] \wedge O_F[R_1[x], F[R_1[x]]] \implies O_F[x, C_1[x, F[R_1[x]]]]) \tag{11}$$

$$\ldots$$

$$(\forall x : I_F[x])(Q_n[x] \wedge O_F[R_n[x], F[R_n[x]]] \implies O_F[x, C_n[x, F[R_n[x]]]]) \tag{12}$$

$$(\forall x : I_F[x])\ (F'[x] = 0) \tag{13}$$

where F' is defined as:

$F'[x] =$ **If** $Q_0[x]$ **then** 0 (14)

elseif $Q_1[x]$ **then** $F'[R_1[x]]$
elseif $Q_2[x]$ **then** $F'[R_2[x]]$
$\ldots$
else $Q_n[x]$ **then** $F'[R_n[x]]$.

Based on this statement we construct a VCG, which takes as an input program (2) annotated with its specification I_F and O_F, and generates the verification conditions (10), (11), (12) and (13). Moreover, the theorem gives, in fact, two statements, namely:

– *Soundness*: If (10), (11), (12) and (13) hold, then the program (2) is correct, and

– *Completeness*: If (2) is correct, then (10), (11), (12) and (13) hold.

A precise proof of the theorem, based on the fixpoint theory of programs [11], is presented in [7], and completed in [9].

3.3 Proving the Verification Conditions

As we have already said, the coherence check is done at the beginning of the verification process—it is also realized by proving the validity of the respective conditions: (3), (4), (5), (6), (7), (8) and (9). Partial correctness is guarantied by (10), (11), (12), and termination—(13).

Proving any of the three kinds of verification conditions has its own difficulty, however, our experience shows that proving coherence is relatively easy, proving partial correctness is more difficult and proving the termination verification condition (it is only one condition) is in general the most difficult one. The latter one is expressed by using a *simplified version*(14) of the initial program (2), and the condition itself expresses a property of that *simplified version* (13). The proof typically needs an induction prover and the induction step may sometimes be difficult to find. Fortunately, due to the specific structure, the proof may be omitted, because different recursive programs may have the same *simplified version*.

Proofs of the verification conditions may be done by using a *Theorema* prover (see, e.g., [3],[5]) or by delivering the proof problem itself to another specialized tool. For serving the termination proofs, actually for omitting the proof redundancy, we are now creating libraries containing *simplified versions* together with their input conditions, whose termination is proven. The proof of the termination may now be skipped if the *simplified version* is already in the library and this membership check is much easier than an induction proof – it only involves matching against simplified versions.

4 Example and Discussion

In order to make clear our experiments, we consider again a powering function P, however we provide this time a different implementation, namely *binary powering*:

$$\begin{aligned} P[x,n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

This program in the context of the theory of real numbers, and in the following formulae, all variables are implicitly assumed to be real. Additional type information (e. g. $n \in \mathbb{N}$) may be explicitly included in some formulae.

The specification is:

$$(\forall x, n : n \in \mathbb{N})\ P[x, n] = x^n. \tag{15}$$

The (automatically generated) conditions for **coherence** are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow \mathbb{T}) \tag{16}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \text{Even}[n]\ \Rightarrow \text{Even}[n]) \tag{17}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\text{Even}[n]\ \Rightarrow \text{Odd}[n]) \tag{18}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \text{Even}[n] \wedge m = (x * x)^{n/2} \Rightarrow \mathbb{T}) \tag{19}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \neg\text{Even}[n] \wedge m = (x * x)^{(n-1)/2}\ \Rightarrow \mathbb{T}) \tag{20}$$

One sees that the formulae (16), (19) and (20) are trivially valid, because we have the logical constant $\mathbb{T}$ at the right side of an implication. The origin of these $\mathbb{T}$ come from the preconditions of the 1 *constant-function-one* and the $*$ *multiplication*.

The formulae (17) and (18) are easy consequences of the elementary theory of reals and naturals. For the further check of **correctness** the generated conditions are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 1 = x^n) \tag{21}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \text{Even}[n]\ \Rightarrow n/2 \in \mathbb{N}) \tag{22}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \text{Even}[n] \wedge m = (x * x)^{n/2}\ \Rightarrow m = x^n) \tag{23}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\text{Even}[n]\ \Rightarrow (n-1)/2 \in \mathbb{N}) \tag{24}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \neg\text{Even}[n] \wedge m = (x * x)^{(n-1)/2}\ \Rightarrow x * m = x^n) \tag{25}$$

$$(\forall x, n : n \in \mathbb{N})\ P'[x, n] = 0, \tag{26}$$

where

$$\begin{aligned} P'[x, n] = \ & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \text{Even}[n] \textbf{ then } P'[x * x, n/2] \\ & \textbf{else } P'[x * x, (n-1)/2]. \end{aligned}$$

The proofs of these verification conditions are straightforward.

Now comes the question: What if the program is not correctly written? Thus, we introduce now a bug. The program P is now almost the same as the previous one, but in the base case (when $n = 0$) the return value is 0.

$$\begin{aligned} P[x, n] = \ & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \text{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

Now, for this buggy version of P we may see that all the respective verification conditions remain the same—and thus the program is correct—except one, namely, (21) is now:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 0 = x^n) \tag{27}$$

which itself reduces to:

$$0 = 1$$

(because we consider a theory where $0^0 = 1$).

Therefore, according to the *completeness* of the method, we conclude that the program P does not satisfy its specification. Moreover, the failed proof gives a hint for "debuging": we need to change the return value in the case $n = 0$ to 1.

Furthermore, in order to demonstrate how a bug might be located, we construct one more "buggy" example where in the "Even" branch of the program we have $P[x, n/2]$ instead of $P[x * x, n/2]$:

$$\begin{aligned} P[x,n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

Now, we may see again that all the respective verification conditions remain the same as in the original one, except one, namely, (23) is now:

$$(\forall x, n : n \in \mathbb{N})\ (\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x)^{n/2}\ \Rightarrow m = x^n) \tag{28}$$

which itself reduces to:

$$m = x^{n/2}\ \Rightarrow m = x^n$$

From here, we see that the "Even" branch of the program is problematic and one should satisfy the implication. The most natural candidate would be:

$$m = (x^2)^{n/2}\ \Rightarrow m = x^n$$

which finally leads to the correct version of P.

5 Conclusions

The approach to program verification presented here is a result of an experimental work with the aim of practical verification of recursive programs. Although the examples presented here appear to be relatively simple, they already demonstrate the usefulness of our approach in the general case. We aim at extending these experiments to industrial-scale examples, which are in fact not more complex from the mathematical point of view. Furthermore we aim at improving the education of future software engineers by exposing them to successful examples of using formal methods (and in particular automated reasoning) for the verification and the debugging of concrete programs.

References

1. Y. Bertot, P. Casteran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer Verlag, 2004.
2. F. Blanqui , S. Hinderer, S. Coupet-Grimal, W Delobel, A. Kroprowski. A Coq library on rewriting and termination. *http://coq.inria.fr/contribs/CoLoR.html*
3. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. In *Journal of Applied Logic, vol. 4, issue 4*, pp. 470–504, 2006.
4. B. Buchberger and F. Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics (in German)*. Springer, 2nd edition, 1981.
5. B. Buchberger, D. Vasaru. Theorema: The Induction Prover over Lists. In *First International Theorema Workshop*, RISC, Hagenberg, Austria, June 1997.
6. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.
7. T. Jebelean, L. Kovács, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2006. To appear.
8. M. Kaufmann and J. S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.
9. L. Kovacs, N. Popov, T. Jebelean. Combining Logic and Algebraic Techniques for Program Verification in Theorema. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, Paphos, Cyprus, 2006.
10. J. Loeckx, K. Sieber. The Foundations of Program Verification. Teubner, second edition, 1987.
11. Manna, Z.: *Mathematical Theory of Computation.* McGraw-Hill Inc. (1974)
12. PVS: Specification and Verification System. *http://pvs.csl.sri.com*

Towards a Static Profiler

Adrian Prantl
Technische Universität Wien

The profiler is an important tool to identify performance bottlenecks in programs. A traditional profiler works by compiling a specially instrumented version of the target program that collects performance data over a series of program executions with typical input data. The collected data is then interpreted and presented to the user as tables and execution graphs.

Since it is up to the user to supply the program with a representative set of input data during the profiling run, it is often left to chance that performance critical paths are triggered during profiling. While this is acceptable for a user interested in the average performance of the program, it is not purposeful for investigating the program's worst-case behaviour; a property especially important to developers of real-time and embedded systems.

In this talk we present the ingredients for a *static profiler* which visualizes an unfolded interprocedural control flow graph by using the results from multiple static analyzers that identify feasible paths and provide upper bounds for loop constructs. The resulting graph is guaranteed to always include the worst-case behaviour of the program.

Our prototype implementation was built using SATIrE program analysis framework as frontend and can analyze C programs. The graph calculation and flow analysis was implemented in Prolog using the Termite library.

References

SATIrE: http://www.complang.tuwien.ac.at/satire/
Termite: http://www.complang.tuwien.ac.at/adrian/termite/

How to Specify the Flow of Data Accessibility: An OO Way of Concurrent Programming

Franz Puntigam

Technische Universität Wien, Institut für Computersprachen
Argentinierstraße 8, 1040 Vienna, Austria
franz@complang.tuwien.ac.at

Abstract. Program structures appropriate for concurrency are often in conflict with object-oriented principles. Especially average programmers need high-level language constructs for concurrency with good integration into the object-oriented paradigm. A key concept in this respect is better static knowledge of the data flow. We propose to explicitly specify the flow of data accessibility in a program. This information is sufficient for a compiler to automatically spawn threads and add synchronization as necessary. Programmers regard the specifications just as assertions.

1 Introduction

It is not easy to express concurrency in a programming language mainly because concurrent units access shared data. We need synchronization to avoid data races and observe data dependences. Thereby we serialize parts of the execution in a statically rather unpredictable way. By spawning concurrent threads we cause the execution to be controlled to a large extent by the data flow – or more precisely the flow of accessibility to shared data – instead of the control flow. Most programs clearly express the control flow, but the flow of data accessibility is rather hidden. That is an important difficulty in concurrent programming.

In the present work we propose a technique to make the flow of data accessibility more visible in an object-oriented language. We want to explicitly express for each method in each object interface

- which shared data are accessible in the method,
- the kind of possible accesses to these data:
  - exclusive read-and-write accesses
  - or consistent read-only accesses (without concurrent write in between)
  - or shared accesses that need synchronization,
- and constraints on values the variables are supposed to hold before method invocation and after return (representing data dependences).

Such information is to a large extent sufficient for a compiler to

- decide where to spawn concurrent threads to ensure a continuous data flow,
- decide where to synchronize threads to avoid data races and comply with data dependences (as enforced by `wait` and `notify` in Java),

- check mutual compatibility of sequential and concurrent program parts,
- and check important properties required for continuous operation.

Programmers can profit from this approach in several ways:

- They write neither code for spawning threads nor for synchronization and still have control over concurrency and can express data dependences.
- The additional information in interfaces conforms to Design by Contract [6] – a concept familiar to object-oriented programmers. Specifications concerning concurrency no longer contradict object-oriented design principles.
- Subtyping considers concurrency according to the substitution principle [5].

We show our approach by examples in Sect. 2, discuss assertion checking in Sect. 3, present rather formal techniques in Sect. 4, address inherent problems in Sect. 5, refer to related work in Sect. 6, and give our conclusions in Sect. 7.

2 How it Works

We demonstrate basics of our approach with an example in a Java-like language:

```
class Buffer {
   public void put(int i) [!full() -> !empty()] { e[top++] = i; }
   public int get() [!empty() -> !full()] { return e[--top]; }
   public boolean empty() [true] { return (top == 0); }
   public boolean full() [true] { return (top == e.length()); }
   public Buffer(int s) [-> !full()] { e = new int[s]; }
   private int e[], top = 0;
}
```

Square brackets on methods represent *access constraints*. We can read expressions in square brackets as assertions (pre- and post-conditions). Accordingly, `put` is invokable only when the buffer is not full and afterwards it is not empty, and `get` only when the buffer is not empty and afterwards it is not full. The arrow between pre- and post-condition indicates state changes of the buffer; an execution of `get` requires exclusive read-and-write access to `this`. Executions of `empty` and `full` need only read-only access as indicated by the availability of an assertion (`true`) and the absence of `->` in the access constraint. A method without any assertion must not access instance variables. The constructor automatically has read-and-write access and ensures a new buffer not to be full.

Access constraints can also be specified for formal parameters and method results. However, there are no post-conditions (expressions to the right of `->`) in access constraints of method results; only conditions to the left of `->` can occur there. Access constraints on local variables, instance variables and class variables are inferred by the compiler. It would be difficult to specify them explicitly because each occurrence of a variable needs its own assertions.

We regard assertions in square brackets to be tokens as used to represent type-states. Tokens are limited resources, and a compiler can statically guarantee that actions depending on tokens are carried out at run time only if these

tokens are available. For example, the compiler can guarantee that `put` is invoked only in a buffer of type `Buffer[!full()->]` without any run-time checks. The associated access constraint implies exclusive access to a non-full buffer. Moreover, tokens can move from one method to another as needed in the computation in a statically predetermined way. In access constraints of methods and formal parameters, tokens to the left of `->` move from the caller to the callee on invocation, and those to the right from the callee to the caller. If a token occurs only on one side, then it flows only in one direction.

Buffers are useful only if several clients access them concurrently and clients compete for exclusive access. We need a further kind of tokens – *cooperative tokens* – to express competing access as shown in the following code snippet:

```
void produce(Buffer[[!full() -> !empty()] ->] p)
  { while(true) { ...; p.put(...); ... }
void consume(Buffer[[!empty() -> !full()] ->] c)
  { while(true) { ...; i = c.get(); ... }
```

There is no exclusive access to the formal parameters `p` and `c` in `produce` and `consume`. Instead, `produce` has to repeatedly acquire a temporary lock on `p` when `p` is not full to get exclusive access, and then perform actions on `p` that cause `p` not to be empty before releasing the lock. An invocation of `put` on `p` changes the state as required. The syntax of the corresponding token (pre-condition on `p`) is the same as the access constraint of `put`. This token moves from the caller to `produce` and remains there because the token occurs only to the left of `->`.[1]

A method like `produce` would be rather useless without another method like `consume` and vice versa. We express this relationship in tokens: If there is a token `[!full() -> !empty()]` in a thread, then there must be at least one further token `[!empty() -> !full()]` in another concurrent thread. In general, for each token [f_1 -> f_2] there must be tokens [f_2 -> f_3], $\ldots$, [f_{n-1} -> f_n] and [f_n -> f_1], eventually all in different threads; the pairwise different assertions $f_1, \ldots, f_n$ must be arrangeable in a cycle. Furthermore, the availability of a cooperative token [f_i -> f_j] is an obligation to repeatedly invoke methods with corresponding access constraints. Otherwise the computation gets stuck with high probability.

The next example shows how the compiler can introduce concurrency:

```
void produce_consume() {
   Buffer b = new Buffer(2); // Buffer[!full()->]
   b.put(1);                 // Buffer[!empty()->]
   int i = b.get();          // Buffer[!full()->]
   produce(b);               // Buffer[[!full() -> !empty()],
                             //        [!empty() -> !full()]]
   consume(b);               // Buffer[[!full() -> !empty()]]
   produce(b);               // Buffer
}
```

[1] The token would move back on return if there was no arrow in the access constraint or the token occurred on both sides of the arrow.

The comments show the types of `b` (including inferred access constraints) after executing the statements on the same lines. First, we construct a new instance of `Buffer` and get the token ensured by the constructor. Because of this token we can invoke `put` on `b` and get another token that allows us to invoke `get`. These statements are executed sequentially without any need of synchronization because all required tokens are available. However, the cooperative token required on the argument of `produce` is not directly available. As discussed in Sect. 4 we can replace an access constraint `[!full()->]` or `[!empty()->]` with `[[!full()->!empty()],[!full()->!empty()],[!empty()->!full()]]`. We need the token required by `produce` twice because `produce` is invoked twice. For the first invocation of `produce` and the invocation of `consume` the compiler has to spawn new threads because these tokens must end up in different threads. To be exact, a new thread is necessary only if the invoked method requires just a single cooperative token on the same parameter; otherwise the invoked method is responsible for spawning the required threads. Each invocation uses up a token until nothing is left.

As shown in this example, thread creation follows a simple algorithm based essentially on the availability of tokens. We expect programmers to be able to understand the algorithm although it is usually advisable to concentrate on the flow of data accessibility instead of the thread structure.

3 Checking Assertions

By regarding assertions as tokens we introduce a token flow that can easily be followed by a compiler. Properties expressed in tokens (except of cooperative tokens) cannot accidentally change because there is no other thread or unknown alias that could modify the object state. However, we have to clarify if a property holds in the first place when introducing a new token or changing a property as in `put` and `get`: After executing the method body we have to dynamically check if the property holds (`!empty()` in `put` and `!full()` in `get` as well as in the constructor). When invoking a method we have to perform a check only in the course of acquiring a lock for a cooperative token.

Dynamic assertion checks can be avoided by using static information accessible by the compiler. We propose a set of static properties and argue that most data dependences needed for synchronization are easily expressible with them. The most important static properties are of the forms $v\!:\!\tau$, $v\!:\!c$ and $v\!:\!c\,..\,c'$ where v is a variable, τ a type, and c and c' are constant values. They are satisfied if v holds a value of type τ, value c, and any value in the range from c to c', respectively. After assigning a value of type τ or a constant value to v the compiler has all needed information; otherwise it will issue an error message. An important advantage of static properties are well-defined relationships between them. For example, we have the following implications: `v:3` $\Rightarrow$ `v:2..4` $\Rightarrow$ `v:1..8` $\Rightarrow$ `v:int`. Of course, we can move a token `v:3` to a method that requires a token `v:2..4`. In contrast, we cannot use a token `empty()` where a token `!full()` is required because the compiler does not know about such relationships.

Static properties of the forms v+i and v−i can be used only as post-conditions together with pre-conditions of the form v:j..j' where i, j, j' are integer constants. They specify that the value of v will be incremented or decremented correspondingly. The next variant of our example shows how we can make use of such static properties:

```
class Buffer8 {
   public void put(int i) [top:0..7 -> top+1] { e[top++] = i; }
   public int get() [top:1..8 -> top-1] { return e[--top]; }
   public Buffer8() [-> top:0] { e = new int[8]; }
   private int e[], top = 0;
}
```

This variant is shorter because we state dependences more directly in terms of variable values. The static determination of the buffer size allows us to compile the following statement sequence:

```
   Buffer8 b = new Buffer8();    // Buffer8[top:0->]
   b.put(1); b.put(2); b.put(3)  // Buffer8[top:3->]
```

The compiler knows that `top` is 3 after three invocations of `put` just from the specifications in the class interface. A corresponding statement sequence would not compile for `Buffer` (instead of `Buffer8`) for several reasons:

- There is no static information about the buffer size. Value 3 can be inappropriate for `top`.
- Unlike `top+1` a post-condition `!empty()` as well as `top:1..8` does not tell us if and how a method invocation modifies `top`.
- There are no invocations of `get` in the code snippet. If there were three invocations of `get`, the compiler could generate concurrent threads so that the buffer size does not matter. No concurrency is necessary for `Buffer8`.

A non-cooperative token specifies two aspects – accessibility (exclusive or consistent read-only) and a property. Accessibility is the more important aspect because with exclusive as well as consistent read-only access we can quite often dynamically determine the property as in the following example:

```
   Buffer[empty()->] emptyTest(Buffer[true ->] b)
      { if (b.empty()) { return b; }; return null; }
```

The parameter `b` has an exclusive pre-condition `true` and no post-condition at all. If possible, `emptyTest` returns this parameter with an exclusive assertion `empty()`; otherwise it returns `null` which can be associated with any assertion in the same way as it can be used instead of an instance of any reference type. For each side-effect-free condition in an `if`-statement depending only on a single parameter (or a single variable or `this`) the compiler can assume the condition to be a valid property of this parameter (or variable or `this`) within the body of `if`. Hence, `b` has the needed property in the `return` statement. As special cases, a condition v `instanceof` τ implies the property v:τ, v==c implies v:c, and v >= c && v <= c' implies v:c..c'. In this context it is not useful to distinguish between static and dynamic properties.

$$a \equiv a \qquad a,b \equiv b,a \qquad a,(b,c) \equiv (a,b),c \qquad a \equiv a,\epsilon \qquad a \equiv a,a$$

$$\frac{a \equiv b \quad b \equiv c}{a \equiv c} \quad \frac{a \equiv b}{b \equiv a} \quad \frac{a \equiv c \quad b \equiv d}{a,b \equiv c,d} \quad \frac{f \Rightarrow g}{f,g \equiv f} \quad \frac{f \Rightarrow g}{f^*,g^* \equiv f^*} \quad \frac{a \equiv c \quad b \equiv d}{\texttt{[}a\texttt{->}b\texttt{]} \equiv \texttt{[}c\texttt{->}d\texttt{]}}$$

Fig. 1. Equivalence of Tokens Sequences

$$\frac{a \equiv b}{a \leq b} \quad \frac{a \leq b \quad b \leq c}{a \leq c} \quad \frac{a \leq c \quad b \leq d}{a,b \leq c,d} \quad f \leq \epsilon \quad f^* \leq \epsilon \quad \texttt{[}f\texttt{->}f\texttt{]} \leq \epsilon$$

$$\frac{f \Rightarrow g}{f \leq g} \quad \frac{f \Rightarrow g}{f^* \leq g^*} \quad \frac{f \Rightarrow f' \quad g' \Rightarrow g}{\texttt{[}f\texttt{->}g\texttt{]} \leq \texttt{[}f'\texttt{->}g'\texttt{]}} \quad \frac{\text{check}(g)}{f \leq f,g} \quad \frac{\text{check}(g)}{f^* \leq f^*,g^*}$$

$$\texttt{[}f\texttt{->}g\texttt{]} \leq \texttt{[}f\texttt{->}f'\texttt{]},\texttt{[}f'\texttt{->}g\texttt{]} \qquad \frac{\text{clean}(f')}{\texttt{[}f\texttt{->}f'\texttt{]},\texttt{[}f'\texttt{->}g\texttt{]} \leq \texttt{[}f\texttt{->}g\texttt{]}}$$

$$\frac{\text{new Proxy}}{f \leq \texttt{[}f\texttt{->}f\texttt{]}} \quad \frac{\text{clean}(f),\text{ no Proxy}}{\texttt{[}f\texttt{->}f\texttt{]} \leq f} \quad \frac{\text{write-lock}(f)}{\texttt{[}f\texttt{->}g\texttt{]} \leq f}\,(^\dagger) \quad \frac{\text{read-lock}(f)}{\texttt{[}f\texttt{->}f\texttt{]} \leq f^*}\,(^\ddagger) \quad \frac{f \Rightarrow g}{f \leq g^*}\,(^\ddagger)$$

Fig. 2. Subsumption of Tokens Sequences

4 Relations on Tokens and Types

Non-cooperative tokens occurring in access constraints with arrows are exclusive tokens, those in access constraints without arrows are read-only tokens. In this section we discuss tokens without their context available. To distinguish between these token kinds we mark read-only tokens with *. For properties f and g, f^* denotes a read-only token, f an exclusive token, and `[`f`->`g`]` a cooperative token.

A *token sequence* is a possibly empty comma-separated list of tokens of the same kind, this is a sequence of exclusive tokens, a sequence of read-only tokens, or a sequence of cooperative tokens. An access constraint is a square bracket containing either a sequence of non-exclusive tokens or two sequences of tokens that are not read-only (separated by an arrow). Equivalence of token sequences is defined in Fig. 1 where f, g denote properties, a, b, c, d token sequences, and ϵ the empty sequence.

Fig. 2 defines a subsumption relation on token sequences. If $a \leq b$ holds, then we can replace token sequence a with token sequence b where a is available and b is required. Most tokens can be removed, that is, replaced by ϵ. However, tokens of the form `[`f`->`g`]` with $f \not\Rightarrow g$ (f does not imply g or f is not known to imply g) must not be removed because such tokens have to be repeatedly used for continuous operation. Many rules in Fig. 2 depend on conditions like implications between properties. The compiler has to guarantee that these conditions are satisfied at run time when applying these rules. For the condition check(g) the compiler has to ensure that g is satisfied, for example, by an `if`-statement. There is a rule allowing the compiler to replace a single cooperative token by two different cooperative tokens. This rule helps us to ensure that properties in cooperative tokens always build cycles. Another rule can reduce the number of

properties in a cycle if no synchronization depends on the withdrawn property f' as expressed by clean(f').

The rules in the last line of Fig. 2 relate tokens of different kinds. Since all tokens in a token sequence must be of the same kind, they must be replaced simultaneously. The compiler has to introduce a new proxy for synchronization (locking) at the position where exclusive tokens are replaced by cooperative tokens. In this proxy the compiler sets exclusive locks when replacing a corresponding cooperative token with an exclusive token and shared read-only locks when replacing a cooperative token with a read-only token. Rules marked with (†) or (‡) are applicable only for a limited amount of time when invoking methods with access constraints corresponding to the tokens at the right-hand-side of $\leq$. On return the tokens at the left-hand-sides of $\leq$ become valid again. Rules marked with (‡) allow the compiler to invoke several such methods simultaneously.

We regard access constraints as annotations of types, no matter whether they are expressed explicitly (for formal parameters and result types) or inferred by the compiler (for local variables, instance variables and class variables). As a consequence, access constraints play an important role in (behavioral) subtyping:

$$\frac{\sigma \leq \tau \quad a \leq b}{\sigma\texttt{[}a\texttt{]} \leq \tau\texttt{[}b\texttt{]}} \qquad \frac{\sigma \leq \tau \quad a \leq b}{\sigma\texttt{[}a\texttt{->]} \leq \tau\texttt{[}b\texttt{]}} \qquad \frac{\sigma \leq \tau \quad a \leq c \quad d \leq b}{\sigma\texttt{[}a\texttt{->}b\texttt{]} \leq \tau\texttt{[}c\texttt{->}d\texttt{]}}$$

Subtyping of types with such annotations resembles subtyping with assertions where a subtype can have weaker pre-conditions (requiring less restrictive tokens) and stronger post-conditions (ensuring more restrictive tokens) than supertypes [5]. Of course, types promising exclusive access to their instances can be used where only read-only access is needed, but not the other way around.

As usual, types of actual parameters must be subtypes of formal parameter types. In the proposed approach, access constraints associated with these types determine to a large extent when to spawn threads and apply synchronization. Most access constraints associated with actual parameters are inferred by the compiler based on the rules in Fig. 2. Essentially, the compiler

– determines all tokens needed by a variable in method invocations (specified in the method signature) as well as tokens becoming available thereby,
– relates tokens becoming available by one invocation with those needed in the next invocation,
– and finally checks for each assignment if all tokens needed by the left-hand-side can be supplied by the right-hand-side.

The rules have to be applied in the last two steps, and as a side-effect of rule applications the compiler possibly has to add code for generating new synchronization proxies, locking, etc. However, the rules are not deterministic: The compiler has some freedom in spawning threads as the following example shows:

```
void test(Buffer[!full() -> !full()] b)
   { b.put(1); b.get(); }
```

It is easy to invoke `put` and `get` sequentially. However, it is also possible to spawn separate threads for `put` and `get` by transforming tokens as follows:

```
Buffer[!full()->]                              (b = new Proxy on b)
  ≤ Buffer[[!full()->!full()]]
  ≤ Buffer[[!full()->!empty()], [!empty()->!full()]]
             (invoke put and get, then ensure clean(!b.empty()))
  ≤ Buffer[[!full()->!full()]]
             (ensure clean(!b.full()) and remove Proxy from b)
  ≤ Buffer[!full()->]
```

The concurrent solution suffers from a large overhead caused by creating and removing a synchronization proxy, acquiring locks, and dynamically ensuring that synchronization no longer depends on `!b.full()` and `!b.empty()`. In each case, `put` and `get` will be executed sequentially.

To avoid such overhead we require from the compiler to introduce cooperative tokens only if there is no solution using non-cooperative tokens. This simple strategy causes the inference of access constraints to become deterministic. Nonetheless, for independent method invocations the rules do not provide enough information to decide between concurrent and sequential execution:

```
void test2(Buffer[!full()->!empty()] b,
           Buffer[!empty()->!full()] c)
   { b.put(1); c.get(); }
```

The two invocations can safely be executed in parallel without any synchronization because they cannot access the same variables. However, it is not clear if parallelism can compensate for the overhead of thread creation in this case.

5 Some Nasty Details

Token-based techniques like the one we use in our approach usually suffer from a number of difficulties. In this section we address the most important ones.

Recursion. Many token-based techniques do not support recursion because we consider methods like `put` to be atomic actions executed in isolation from other methods. An invocation of `put` within `put` could occur in an inconsistent state and thereby compromise isolation. Programmers usually prefer recursion over isolation. Therefore, the approach proposed in this paper supports recursion in the sequential as well as concurrent case. Technically speaking, by synchronization we can convert a cooperative token into an exclusive one that can be used without further synchronization, for example, in a recursive invocation. It is also possible to have several concurrent recursive invocations by introducing a further synchronization proxy and replacing the exclusive token with cooperative tokens. There can simultaneously exist any number of synchronization proxies on the same object, all but at most one of them with an exclusive lock. By the use of assertions as tokens we still can ensure the required isolation. Pre-conditions have to be satisfied also for recursive invocations.

Token Loss. Especially in exceptional cases we easily lose tokens. For example, we lose a token moved into a method if the method terminates with an unexpected exception. We can dynamically reconstruct the property expressed by the assertion of a token, but we cannot reconstruct the accessibility information by hand because usually we do not know anything about other threads that may have got the token from the exceptionally terminated method. Being careless we can lose tokens even in the normal control flow – for example, tokens in access constraints of instance variables belonging to objects ready to be garbage collected or simply being untraceable.

Token loss is caused by many factors to be addressed differently. First of all we have to organize the software so that we do not lose still needed tokens in the normal control flow. For example, we put object references together with corresponding tokens into a repository where users can easily find them together on demand; see the notes on arrays and collections. Expected exceptions can carry object references together with tokens. For unexpected exceptions thrown by the run-time system we have no satisfactory solution, but several approaches:

- The exception handling mechanism creates a repository containing all object references (together with their tokens) available as local variables where the exception occurred. The exception handler is responsible for exploring this repository and making use of needed tokens. This approach causes an overhead for throwing as well as handling exceptions, and tokens associated with instance variables at the time when the exception occurred will probably be untraceable by the exception handler.
- When a method terminates with an exception, the exception handling mechanism automatically returns all accessibility information provided by the caller on invocation (but all corresponding assertions will simply be `true` for exclusive tokens). If values have been assigned to variables annotated with tokens to be returned, the exception handling mechanism has to write `null` to these variables, and if threads have been spawned and locks acquired because of such tokens, then the threads must be terminated and the locks reset. This approach allows for simple implementations of exception handlers, but it is probably difficult to understand which variables will be set to `null`.
- Similar as above, the exception handling mechanism returns accessibility information, but only for tokens provided by the caller *and* to be returned to the caller according to the normal control flow. In this approach, the exception handling mechanism has to set only those variables to `null` that would contain another value or would not be annotated with this token at normal termination of the method. However, some tokens can be lost.

Stopping Continuous Operation. The support of continuously operating systems through cooperative tokens is a main feature of the proposed approach as the `produce_consume` example shows [13]. As long as there is a producer there must also be a consumer and vice versa. Once the continuous operation has been started, the token needed by the consumer cannot come together with the token

needed by the consumer, and hence no rule in Fig. 2 is applicable to combine these tokens again. In the normal case we cannot stop the producer independently from the consumer. To stop the producer (or any continuous operation in general) in exceptional cases we introduce a *stop mode.* After entering stop mode (by invoking a specific method) it is no longer possible to get a lock on the corresponding synchronization proxy. All threads waiting for a lock will get an exception instead. There is no way back from stop mode.

Arrays and Collections. Each occurrence of a variable has its own access constraint computed by the compiler. For arrays and other collections of items the access constraint can differ for each item. Since the compiler does not know array indexes in general, it is impossible to statically compute different access constraints for different indexes. Similar problems arise for other collections of items. There are two ways to solve these problems:

- Each item in the collection is associated with the same access constraint. This approach works quite well if always the same operation is applied to all items in a collection. Otherwise we have to assume that a token is removed from each item if it is removed from one item, and we lose tokens added to only one item (but not all items).
- Each item in a collection is associated with its own access constraint, and these access constraints are managed dynamically. This approach is more flexible and powerful, but access constraints are no longer a purely static concept and tokens are run-time entities. Explicit dynamic checks of the availability of tokens can be helpful (where brackets on `a` denote an array, those on `Buffer` access constraints):

```
Buffer a[] = ...;
if (a[i] instanceof Buffer[!empty()->]) { a[i].get(); }
```

There is an appropriate combination of both approaches: The compiler statically computes tokens available for all items, and the run-time system stores additional tokens of some items in the collection. Tokens computed by the compiler are directly usable while explicit dynamic checks are necessary before using additional tokens. As a generalization it is useful to associate additional tokens to all variables with a corresponding declaration. The additional tokens are accessible through type checks with `instanceof` and through type casts.

6 Related and Future Work

The proposed approach has similarities with the SCOOP model of concurrency [7]. Both, the SCOOP model and our model, use assertions and disclose information on variables for synchronization. However, the way how and the time when such information becomes available to clients is different: In the SCOOP model every client can get access to corresponding variables at run time while in our model much information is statically available and usually only few clients

can access the variables at run time. Tokens give additional information that allows the compiler to automatically spawn and synchronize concurrent threads.

There are many approaches to ensure unique access [3], most of them by avoiding aliases. The token-based approach used in the present work has been developed from a process type model [9–11] for active objects – essentially an object-oriented variant of linear types [4]. This concept restricts the way how to access objects without preventing aliasing [12, 13].

Synchronization based on tokens has a long tradition: Petri Nets have been explored for nearly half of a century as a basis of synchronization [8]. In general, expressing states by abstract tokens has clear (both practical and theoretical) advantages over expressing them more concretely by values in instance variables: Tokens are much easier tractable than concrete states especially when used in a static analysis. Many proposals use tokens to express abstract object states [1, 2, 14]. In our approach we combine abstract tokens with concrete variable values to get more flexibility.

Many programming languages consider concurrency to be orthogonal to the object model. By the use of object accessibility and assertions as tokens we exploit inherent relationships between objects as a basis for concurrency and synchronization. Concurrency is subordinate to object-oriented factorization. Other than in the concurrency model in Java, a thread that acquires a lock in our approach is in no way privileged compared to other threads because only the availability of tokens counts, not the thread identity. This property is an important step towards modularity of synchronization. A further step is the support of recursion while still executing methods atomically and in isolation. Since the compiler deals with concurrency on a class by class basis, concurrency is almost completely modular. However, there remains a small non-modular rest: To prevent deadlocks the compiler needs global information [13]. In this respect our approach does not differ from many other approaches.

The present work is in an early stage. Currently there is no implementation, and no practical evaluation of this approach has been carried out so far. In future work we want to address (among others) the following topics:

- The compiler has some freedom to decide between sequential and concurrent execution. We need appropriate heuristics to make a beneficial decision.
- Our proposal provides a foundation for assured continuous operation [13]. More work is needed to strengthen the guarantees given by the compiler. Deadlock prevention in at least some cases is one of the goals.
- We need more advanced concepts and more experience to deal with time as well as with exceptional cases (like exception handling and stop mode) in a practical setting.

7 Conclusions

We have explored a way to express the flow of data accessibility in object interfaces so that a compiler can use this information to execute methods in concurrent threads and provide synchronization between them. Based on this in-

formation the compiler can always determine if it is necessary or unfeasible to spawn concurrent threads; for independent computations further information is necessary to decide if it is advantageous to do so. Programmers specify the flow of data accessibility essentially in the form of assertions that are considered to be tokens. As a consequence, object-oriented design principles dominate the program structure, not concurrency. The approach supports subtyping and ensures to some extent continuous operation of the system.

References

1. Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *ESEC/FSE-13*, pages 217–226, Lisbon, Portugal, September 2005. ACM Press.
2. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP 2004 – Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, Oslo, Norway, June 2004. Springer-Verlag.
3. Sophia Drossopoulou, David Clarke, and James Noble. Types for hierarchic shapes. In *ESOP*, pages 1–6, 2006.
4. Naoki Kobayashi, Benjamin Pierce, and David Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.
5. Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. Proceedings OOPSLA'93.
6. Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.
7. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition edition, 1997.
8. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
9. Franz Puntigam. Type specifications with processes. In *Proceedings FORTE'95*, Montreal, Canada, October 1995. IFIP WG 6.1, Chapman & Hall.
10. Franz Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–388, Jyväskylä, Finland, June 1997. Springer-Verlag.
11. Franz Puntigam. *Concurrent Object-Oriented Programming with Process Types.* Der Andere Verlag, Osnabrück, Germany, 2000.
12. Franz Puntigam. See the pet in the beast: How to limit effects of aliasing. In *International Workshop on Aliasing, Confinment and Ownership in object-oriented programming (IWACO 2007)*, Berlin, Germany, July 2007.
13. Franz Puntigam. Synchronization as a special case of access control. *Electr. Notes Theor. Comput. Sci.*, 241:113–133, 2009.
14. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

Towards a Parallel Search for Solutions of Non-deterministic Computations

Fabian Reck and Sebastian Fischer

Department of Computer Science
Christian-Albrechts-University of Kiel
D-24118 Kiel
{fre,sebf}@informatik.uni-kiel.de

Non-determinism is one of the distinctive built-in features of logic and functional logic programming languages such as Prolog and Curry. In other programming languages non-determinism can be modeled using apropriate abstractions. In Haskell this is usually done by means of non-determinism monads. Since the different results of a non-deterministic computation do not depend on each other it should be possible to compute them in parallel. In this paper we explore different possibilities to execute non-deterministic Haskell programs in parallel. Together with our latest attempts to compile Curry programs to monadic Haskell we already achieve preliminary but encouraging results.

We present three different implementation of a parallel execution of non-deterministic monadic Haskell programs:

1. dividing the computation into a fixed number of sub-computations and assign each of them to a different thread,
2. using a Bag-of-Tasks approach to distribute small tasks to a fixed number of threads, and
3. using the `par`-combinator that is implemented in the GHC to give *hints* to the Haskell runtime system about which parts of the computation can be done in parallel.

A detailed description of these approaches together with a discussion of their properties and selected benchmarks is published in the proceedings of the 39th annual conference of German Gesellschaft für Informatik [1].

References

1. Reck, F., Fischer, S.: Towards a Parallel Search for Solutions of Non-deterministic Computations. In: INFORMATIK 2009, Im Focus das Leben, 39. Jahrestagung der Gesellschaft für Informatik. LNI, GI (2009)

Rekursionspräzise Intervallanalysen

Dirk Richter (richterd@informatik.uni-halle.de)

Martin-Luther-Universität Halle-Wittenberg

Zusammenfassung Intervallanalysen bestimmen zu einem gegebenen Programm konservative oder nicht-konservative Wertebereiche von Variablen. Desto genauer diese Wertebereiche bestimmt werden können, desto präziser sind die darauf basierenden Analysen wie z.B. die Laufzeitschätzung von Programmen bzw. führen zu weniger Fehlalarmen wie z.B. bei der Prüfung auf Feldzugriffe außerhalb zulässiger Indizees (ArrayOutOfBounds). Bei **un**beschränkter Rekursion und **un**beschränktem Speicher (Heap/Halde) ist das Problem der Bestimmung von Wertebereichen maximaler Genauigkeit (auch als exakte Wertebereiche bezeichnet) **unentscheidbar**. Im Folgenden wird gezeigt, wie bei **un**beschränkter Rekursion und beschränktem Speicher derartige exakte Wertebereiche in polynomieller Zeit (bzgl. Modellgröße) bestimmt werden können.

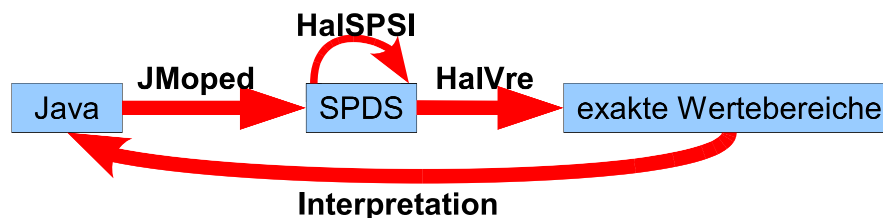
Schlüsselworte: exakte Wertebereichsanalyse, Intervallmenge, SPDS

1 Einleitung

Da viele Programmanalysen i.A. unentscheidbar für die meisten Hochsprachen sind, muss für die Analyse von der Semantik eines Programms abstrahiert werden. So ist es möglich, bestimmte Eigenschaften über das Verhalten eines Programms statisch zu bestimmen. Durch diese Abstraktion entstehen naturgemäß Ungenauigkeiten in den Analyseergebnissen. Ungenaue Analyseergebnisse führen aber in der Praxis zu ungenaueren Aussagen z.B. über die erwartete Laufzeit von Programmen oder zu unnötig großen Chip-Designs. Man ist daher bestebt, diese Ungenauigkeiten in den Analyseergebnissen zu minimieren. Eine Möglichkeit dazu bietet die korrekte Abbildung von Methodenaufrufen und Rekursion auf sog. symbolische Kellersysteme (SPDS). Die Beschränkung auf eine maximale Anzahl an Methodenaufrufen ist dann nicht nötig und vermeidet eine exponentielle Modellvergrößerung z.B. durch Inlining. Die hier betrachteten SPDS unterstützen neben einem Laufzeitkeller, Berücksichtigung von Rekursion, Prozedurparametern, lokalen und globalen Variablen allerdings **keine** Referenz-Paramter beim Methodenaufruf, da sie andernfalls turingmächtig wären und somit viele Analysen wegen dem Halteproblem unentscheidbar werden würden [1].

Programme (in unseren Experimenten Java) werden in einem ersten Schritt in solche SPDS (Rekursionsmodell) überführt (siehe Abbildung 1), welche präzise das rekursive Verhalten des Programms beschreiben. Für diese Rekursionsmodelle können, wie in Abschnitt 4 gezeigt, exakte Wertebereiche berechnet

Abbildung 1. Vorgehen beim Bestimmen der Wertebereiche von Variablen



werden. Je nach Art der Modellgenerierung lassen diese exakten Wertebereiche im Modell Rückschlüsse auf die Wertebereiche des ursprünglichen Programms zu (Abschnitt 5). Da die Bestimmung exakter Wertebereiche (nicht nur komplexitätstheoretisch) aufwändig ist, sind möglichst kleine Modelle für die Durchführbarkeit entscheidend. Derartige Verkleinerung von SPDS ist Gegenstand früherer Veröffentlichungen [2–5]. In diesen Veröffentlichungen wurde unser Werkzeug **HalSPSI** (Halle's Symbolic Pushdown System Improver) beschrieben, welches als Source-To-Source-Compiler gegebene SPDS für die Zwecke einer effizienteren Modellprüfung verbessert bzw. die Modellprüfung überhaupt erst ermöglicht. Gleichsam verbessert dieses Werkzeug die Anwendung von rekursionspräzisen Intervallanalysen (siehe Abschnitt 8).

Informationen über das Modellverhalten können z.B. durch im Programmiersprachenumfeld gängige Programmanalysen gewonnen werden. In den hier betrachteten Fällen werden für jede Variable an jeder Marke bzw. symbolischen Konfiguration[1] verschiedene Eigenschaften berechnet. Bei einer sog. Intervallanalyse werden z.B. obere und untere Schranken für die zur Laufzeit realisierbaren Variablenwerte bestimmt (siehe Abschnitt 3). Dies gelingt etwa durch eine Fixpunktberechnung mittels Transferfunktionen [6, 7] oder durch Modellierung und Lösen eines linaren Programms [8]. Im Folgendem soll eine weitere Möglichkeit (implementiert in unserem Tool Halle's Variable Range Extractor **HalVre**, siehe Abbildung 1) zur Bestimmung solcher Variablenwerte vorgestellt und mit einem eher traditionell auf Fixpunktberechnung basierendem Verfahren verglichen werden. So ist es unter Berücksichtigung von beliebigen Methodenaufrufen und Rekursion statisch möglich, Wertebereiche von Variablen vorherzusagen.

In [2] und [3] wurde gezeigt, dass verschiedene Modellanalysen im Gegensatz zur Anwendung bei herkömmlichen Programmiersprachen **entscheidbar** werden für SPDS (siehe auch [1]), was prinzipiell die Existenz sog. **exakter** Modellanalyseverfahren für SPDS nachweist. Dabei heißt eine Modellanalyse dann **exakt**, wenn das Analyseergebnis weder eine Über- noch eine Unterapproximation darstellt, also präzise das Verhalten des Modells berücksichtigt. So führt die Verknüpfung von einer Konstantenpropagation und Konstantenfaltung sowie Copy-Propagation [9] zu einer nicht exakten (konservativen) Äquivalenzanalyse [5]. Bei dieser kann es Variablen geben, die zwar gleich oder gar konstant sind,

[1] im Sinn von Programmiersprachen auch als ein Programmpunkt auffassbar

Listing 1.1. Java Beispiel zur Berechnung der Fakultät

```
int fac(int n) {
    if (n<=1) return 1;
    return n*fac(n-1);
}
```

dies aber nicht durch die Analyse entdeckt wird. Eine solche Analyse heißt **konservativ**, falls zudem jede durch die Analyse vorhergesagte Äquivalenz auch in jeder Simulation bzw. Ausführung auftritt. Eine exakte Äquivalenzanalyse für SPDS ist in [5] beschrieben. Äquivalenzanalysen und Intervallanalysen haben gemeinsam, dass sie beide in der Lage sind, konstante Variablen zu erkennen (siehe Abschnitt 3 und Listing 1.3). Werden beide Analysen exakt durchgeführt, so erkennen sie natürlich auch genau die gleichen Konstanten.

2 Rekursionsmodelle

$M = (S, \rightarrow, L_A)$ heißt **Kripkestruktur**, falls S und A (nicht notwendigerweise endliche) Mengen sind, $\rightarrow \subseteq S \times S$ und $L_A : S \rightarrow 2^A$. Zur Beschreibung von (unendlich) großen Kripkestrukturen können Kellersysteme (Pushdown Systems) verwendet werden. $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ heißt **Kellersystem**, falls P eine Menge von Zuständen, Γ eine endliche Menge (Kelleralphabet) und $\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine Menge von Transitionen ist. Informell ist ein Kellersystem ein Kellerautomat ohne Eingabe. Solche Kellersysteme beschreiben einen sehr großen sog. Zustandsraum, was zu dem aus der Modellprüfung bekannten Problem der Zustandsraumexplosion führt. Da Kellersysteme ihrerseit per Definition über Zustände verfügen, bezeichnen wir diesen Zustandsraum als Konfigurationenraum, wobei (p, v) **Konfiguration** heißt, falls $p \in P$ und $v \in \Gamma^*$. Eine Konfiguration (p, v) heißt **erreichbar**, falls es beginnend aus einer der Initialkonfigurationen eine Folge von Transitionen nach (p, v) gibt (ein Berechnungspfad, oder kurz **Pfad**). (p, a) heißt **Kopf** der Konfiguration (p, aw), falls $a \in \Gamma$ und $w \in \Gamma^*$. Bei der Berechnung der rekursionspräzisen Intervallanalyse wird der Umstand ausgenutzt, dass die Menge der Köpfe endlich ist, auch wenn der Konfigurationenraum unendlich sein kann. Auf Konfigurationen wird die Transitionsrelation $\hookrightarrow$ erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$. Mit $\xrightarrow{*}$ wird die reflexive und transitive Hülle von $\rightarrow$ bezeichnet. Bei einem **Symbolischen Kellersystem** (**SPDS**) werden die Transitionen nur indirekt (symbolisch) mittels Relationen beschrieben, was die Angabe des vollständigen Kellersystems vereinfacht [10]. Im Folgenden wird ein solches symbolisches Kellersystem als **Rekursionsmodell** bezeichnet, wenn es aus einer höheren Programmiersprache wie Java oder C mit unbeschränkten Methodenaufrufen gewonnen wurde.

Wie in den Listings 1.1 und 1.2 zu sehen, können SPDS mit Hilfe der Modellsprache Remopla [11] sehr kompakt beschrieben werden. Dabei wurden 16 Bit (angehängt an eine Variablendeklaration) zur Modellierung von Ganzzahlen

Listing 1.2. Mit JMoped transformiertes Remopla Beispiel von Listing 1.1

```
1   module int fac(int v0(16))
2   {
3   int s0(16);
4   int s1(16);
5   int s2(16);
6   fac_I:    s0=v0, s1=s0, s2=s1;
7   fac_I1:   s0=1, s1=s0, s2=s1;
8   fac_I2:   if
9             :: s1>s0 -> goto fac_I7, s0=s2;
10            :: else -> s0=s2;
11            fi;
12  fac_I5:   s0=1, s1=s0, s2=s1;
13  fac_I6:   return s0;
14  fac_I7:   s0=v0, s1=s0, s2=s1;
15  fac_I8:   s0=v0, s1=s0, s2=s1;
16  fac_I9:   s0=1, s1=s0, s2=s1;
17  fac_I10:  if
18            :: s1>=s0 -> s0=s1-s0, s1=s2;
19            :: else -> s0=(65536-(s0-s1)%65536)%65536, s1=s2;
20            fi;
21  fac_I11:  s0=fac_I(s0);
22  fac_I14:  s0=(s1*s0)%65536, s1=s2;
23  fac_I15:  return s0;
24  }
```

(Integern) verwendet, um das Verhalten des Java Programms nachzubilden. Remopla ist zwar syntaktisch ähnlich zu Promela (Eingabesprache für den SPIN Modellprüfer), unterstützt aber keine parallelen Prozesse, dafür synchron parallele Konfigurationenübergänge und exakte Rekursion. Exakte Rekursion bedeutet hier, dass die in einem Programm enthaltenen Methodenaufrufe durch das Modell weder unter- noch überapproximiert werden, sondern analog dem Laufzeitsystem moderner Programmiersprachen in einem Keller verwaltet werden.

Neben lokalen Variablen loc_q und Parametern $pars_q \subseteq loc_q$ eines Moduls q (auch Prozedur genannt) können in Remopla auch globale Variablen *globs* sowie Arrays deklariert werden. Die lokalen Variablen und Methodenparameter werden in SPDS als Bitvektoren über dem Kelleralphabet zusammen mit der Aufrufhierarchie im Keller repräsentiert. Globale Variablen (in Java Klassenvariablen), die Halde (Heap) sowie Ausnahmen (Exceptions) werden mit Hilfe der Zustände eines Kellersystems beschrieben[2].

Formal kann Remopla wie folgt zusammengefasst werden. Seien $Vars_q := globs \cup loc_q$, $EXPR_{Vars_q}$ arithmetische Ausdrücke über diesen Variablen und

[2] Ziel früherer Arbeiten [2–5] war es, das dazu nötige Kelleralphabet und die benötigten Kellerzustände bereits symbolisch a priori zu verringern.

$\Phi^q : Vars_q \rightarrow \mathbb{N}$ eine Variablenbelegung, welche aus dem Kopf einer Konfiguration bestimmt wird, sowie $[\![e]\!]_{\Phi^q}$ die Auswertung eines Ausdrucks $e \in EXPR_{Vars_q}$ mittels der Variablenbelegung Φ^q. Dann sind die wichtigsten Remopla-Anweisungen:

– $\mathbf{x}_1 = \mathbf{e}_1,\ \mathbf{x}_2 = \mathbf{e}_2, \ldots,\ \mathbf{x}_n = \mathbf{e}_n$**;** mit $x_i \in Vars_q$ und $e_i \in EXPR_{Vars_q}$ ein synchron paralleler Konfigurationenübergang, welcher jeder Variablen x_i den ausgewerteten Ausdruck $[\![e_i]\!]_{\Phi^q}$ zuweist.
– $\mathbf{p}(\mathbf{e}_1,\ \mathbf{e}_2, \ldots,\ \mathbf{e}_n)$**;** mit $e_i \in EXPR_{Vars_q}$ ein Modulaufruf an das Modul p mit Call-By-Value Semantik.
– **return e;** mit $e \in EXPR_{Vars_q}$ ein Modulende mit Rückgabewert $[\![e]\!]_{\Phi^q}$.
– **goto L;** ein unbedingter Sprung an die Marke L.
– **if :: b**$_1$ **− > s**$_1$**; :: b**$_2$ **− > s**$_2$**; . . . :: b**$_n$ **− > s**$_n$**; fi;** eine bedingte Anweisung mit $b_i \in EXPR_{Vars_q}$ und $[\![b_i]\!]_{\Phi^q} \in \{0, 1\}$, die eine zufällig ausgewählte Anweisung s_k mit $[\![b_k]\!]_{\Phi^q} = 1$ ausführt.

Kommentare gelten bis zum Zeilenende und werden mit dem Symbol # eingeleitet. Für Details zur Konstruktion von Remopla-Modellen aus C- und Java-Programmen sei auf [12–14, 2] verwiesen.

3 Konservative Wertebereichsanalyse für SPDS

In userem Werkzeug **HalSPSI** wurde eine sog. konservative Wertebereichsanalyse durch Kombination von *Range Propagation* und *Range Analysis* [15] basierend auf einer Fixpunktberechnung umgesetzt. Sie ist interprozedural[3] sowie flusssensitivt[4]. Konservativ bedeutet hier, dass nicht für jeden durch **HalSPSI** vorgergesagter Wert einer Variablen es einen Simulationslauf des SPDS geben muss. Wird aber ein gewisser Wert einer Variablen bei einem Simulationslauf angenommen[5], so ist dieser in dem konservativen Analyseergebnis enthalten. Techniken wie Aufweitung (*Widening*) und Einengung (*Narrowing*) [16, 7] können zur Konvergenzverbesserung der Fixpunktberechnung eingesetzt werden und tragen zu einer Verstärkung der Ungenauigkeit der Analyse bei. Da ihr Einsatz für SPDS nicht zwingend erforderlich ist (siehe Abschnitt 7), wurde aus Präzisionsgründen in **HalSPSI** darauf verzichtet. Ziel von Intervallanalysen ist die Bestimmung von oberen und unteren Schranken für die zur Laufzeit realisierbaren Wertebereiche. In der in **HalSPSI** implementierten Wertebereichsanalyse wird nicht nur eine solche obere und untere Schranke je Variable bestimmt, sondern vielmehr der ganze potentielle Werteraum (alle möglichen konkreten Variablenbelegungen) durch eine Vielzahl von Intervallen je Variable und Programmpunkt abgebildet. Eine obere und untere Schranke kann daraus abgeleitet werden.

Durch diese Eigenschaften wird eine Präzisionssteigerung der Analyse gegenüber konventioneller Intervallanalysen erzielt, welche lediglich **eine** obere und untere Schranke beachten.

[3] Im Gegensatz zu intraprozeduralen Analysen (nur innerhalb von Prozeduren) werden hier Analyseergebnisse über Prozedurgrenzen transportiert.
[4] Im Gegensatz zu flussinsensitiven Analysen wird hier die zeitliche Abfolge von Anweisungen berücksichtigt.
[5] Dieser wird im Folgendem auch kurz als **realisierbar** bezeichnet.

Abbildung 2. Beispiele zur Operatorinterpretation bei Intervallmengen

```
{(3,1000)} << {(1)} = {(6), (8), (10), (12), ..., (2000)}
{(3,1000)} * {(3,1000)} = {(9), (12), (15, 16), (18), ..., (1000000)}
{(1000)} / {(3,70)} = {(142), (144), (147), (149), ... ,(3333)}
{(1000)} >> {(0,10)) = {(0,1), (3), (7), (15), ..., (1000)}
{(1001)} % {(10,100)} = {(0, 2), (5) ,(7) ,(9), ... (77), (81)}
```

Listing 1.3. Mit Konstantenfaltung/-Propagation nicht erkennbare Konstante y

```
L: y = x/2;
if
  :: (x = 0) -> x = 1; goto L;
  :: else -> break;
fi
```

Im Folgenden schreiben wir $x@l \in \{(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\}$, wenn die Analyse bestimmt hat, dass die Variable x an der Konfiguration l in einem der Intervalle $[a_i, b_i]$ liegt. Wir bezeichnen $\{(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\}$ als **Intervallmenge** und schreiben statt (a, b) auch (a), falls $a = b$. Um arithmetische Operationen möglichst präzise abstrakt zu interpretieren, müssen diese auf derartige Intervallmengen verallgemeinert werden. Bei den Operationen $*(Multiplikation), /(Division), \%(Division\ mit\ Rest), << (Shift\ links)$ als auch $>>$ $(Shift\ rechts)$ können die Intervalle ggf. in einzelne Werte „zerfallen“, wie die Beispiele aus Abbildung 2 zeigen. Insbesodere das zweite Beispiel erzeugt bereits über 173000 Intervalle. Dies ist vermutlich einer der Gründe, weswegen in Intervallanalysen in der Litertur gern von solchen Rechenoperationen abstrahiert wird [17]. **HalSPSI** hingegen (wie auch **HalVre**) interpretiert diese Operationen vollständig, da dies in der Praxis selten vorkommende Artefakte sind und sich zudem die Anzahl der Intervalle auf die in der Tabelle 1 angegebene Anzahl beschänken. Vielmehr steigt statt dessen die gewonnene Präzision.

Tabelle 1. Maximale Anzahl an Intervalle verschiedener Bit-Größen

Integer-Bitbreite	2	3	4	5	6	7	8	9	10	14	16
Max. Intervalle	2	4	8	16	32	64	128	256	512	8192	32768

Wie Listing 1.3 zeigt, können durch Intervallanalysen auch Konstanten identifiziert werden, welche mit anderen einfacheren Methoden nicht erkannt werden. Die Variable y hat im Listing 1.3 stets den Wert 0, was bereits durch eine ein-

fache Intervallanalyse erkannt wird, nicht jedoch durch Konstantenfaltung oder -Propagation.

Durch Kontextsensitivität wie z.B. durch Context-Cloning [18] (Vorsicht: Modellvergrößerung) oder Berücksichtigung von Call-Strings [19, 20] kann die Präzision der konservativen Wertebereichsanalye bezüglich der Rekursion weiter verbessert werden. Vollständige Kontextsensitivität jedoch ohne Modellvergrößerung kann durch exakte Wertebereichsanalysen erreicht werden, weshalb im Folgenden eine solche beschrieben werden soll.

4 Exakte Wertebereichsanalyse für SPDS

Im Folgenden wird kurz beschrieben, wie unser Werkzeug **HalVre** exakte Intervallmengen für SPDS bestimmt.

Lemma 1. ***(Entscheidbarkeit der Intervallanalyse)***
Seien $a, b \in \mathbb{R}$. Dann ist für SPDS ***entscheidbar****, ob für eine Variable x an einer erreichbaren Konfiguration p stets gilt: $a \leq x \leq b$.*

Proof. Reduktion auf Erreichbarkeitsproblem[6] (analog Äquivalenzanalyse in [3]).

Theorem 1. ***(Entscheidbarkeit der Wertebereichsanalyse)***
Seien $a_i, b_i \in \mathbb{R}$ mit $i \in \{1, 2, \ldots, n\}$. Dann ist für SPDS ***entscheidbar****, ob für eine Variable x an einer erreichbaren Konfiguration p stets gilt: $\exists i : a_i \leq x \leq b_i$.*

Proof. trivial nach Lemma 1.

Auf Grund dieser Entscheidbarkeitsaussage existiert eine kleinste Intervallmenge, welche in diesem Sinne die wenigsten Variablenwerte überdeckt, dennoch konservativ ist und von jeder exakten Wertebereichsanalyse berechnet wird. So sind exakte Intervallanalysen wie auch exakte Wertebereichsanalysen stets automatisch interprozedural, kontextsensitiv, flusssensitiv und pfadsensitiv etc..

Theorem 2. ***(Komplexität exakter Wertebereichsanalysen)***
Eine exakte Wertebereichsanalyse für ein SPDS ist komplexitättheoretisch mindestens so schwer wie die Erreichbarkeitsprüfung von Konfigurationen[7].

Proof. Informal: sie löst die Erreichbarkeitsfrage für alle Marken des SPDS. □

In [10] wird beschrieben, wie zu einem gegebenen SPDS P symbolisch ein endlicher Automat $Post^*(P)$ konstruiert werden kann, welcher die Menge aller erreichbaren Konfigurationen aus gegebenen Initialkonfigurationen akzeptiert. Mittels $Post^*(P)$ kann zu jeder SPDS Marke q eine charakteristische Funktion $f^q : \{0, 1\}^* \to \{0, 1\}$ für die exakten Variablenbelegungen für lokale und globale

[6] Die Frage, ob es im SPDS einen Pfad (Transitionenfolge) von einer der Anfangskonfigurationen zu einer gegebenen Konfiguration, Kopf oder Marke gibt.

[7] Modellprüfung

Listing 1.4. Definierte Variablen a_i mit Dimensionen d_i und Bitbreiten k_i

```
int a1[d1](k1);
int a2[d2](k2);
...
int am[dm](km);
```

Variablen[8] bestimmt werden durch Beschränkung der Transitionen aus $Post^*(P)$ auf Köpfe (Schritt 1). Aus diesen Köpfen können dann in einem zweiten Schritt die entsprechenden Wertebereiche der Variablen ermittelt werden.

Zur Veranschaulichung seien nun in einer Methode P an der Marke q die lokalen und globalen Variablen a_i (einschließlich Parameter) wie in Listing 1.4 gegeben mit Dimensionen d_i (a_i ist ein Array gdw. $d_i > 1$) und Bitbreiten k_i:

Dann ist mit $x_{i,d,k} \in \{0,1\}$ für $i \in \{1,2,\dots m\}, d \in \{1,2,\dots d_i\}, k \in \{1,2,\dots k_i\}$

$$
\begin{array}{llllll}
f^q(& x_{1,1,1}, & x_{1,1,2}, & x_{1,1,3}, & \dots, & x_{1,1,k_1}, \\
& x_{1,2,1}, & x_{1,2,2}, & x_{1,2,3}, & \dots, & x_{1,2,k_1}, \\
& \dots & & & & \\
& x_{1,d_1,1}, & x_{1,d_1,2}, & x_{1,d_1,3}, & \dots, & x_{1,d_1,k_1}, \\
& x_{2,1,1}, & x_{2,1,2}, & x_{2,1,3}, & \dots, & x_{2,1,k_2}, \\
& x_{2,2,1}, & x_{2,2,2}, & x_{2,2,3}, & \dots, & x_{2,2,k_2}, \\
& \dots & & & & \\
& x_{2,d_2,1}, & x_{2,d_2,2}, & x_{2,d_2,3}, & \dots, & x_{2,d_2,k_2}, \\
& \dots & & & & \\
& \dots & & & & \\
& x_{m-1,d_{m-1},1}, & x_{m-1,d_{m-1},2}, & x_{m-1,d_{m-1},3}, & \dots, & x_{m-1,d_{m-1},k_{m-1}}, \\
& x_{m,1,1}, & x_{m,1,2}, & x_{m,1,3}, & \dots, & x_{m,1,k_m}, \\
& x_{m,2,1}, & x_{m,2,2}, & x_{m,2,3}, & \dots, & x_{m,2,k_m}, \\
& \dots & & & & \\
& x_{m,d_m,1}, & x_{m,d_m,2}, & x_{m,d_m,3}, & \dots, & x_{m,d_m,k_m}) \quad = 1
\end{array}
\tag{1}
$$

gdw. es eine **realisierbare** Wertbelegung Φ an der Marke q gibt, so dass $\forall i \in \{1,2,\dots,m\}, \forall d \in \{0,2,\dots,d_i - 1\}$:

$$\Phi(a_i[d]) = \sum_{b=1}^{k_i} x_{i,d,b} \cdot 2^{k_i - b}. \tag{2}$$

Φ beschreibt somit **einen** realisierbaren Kopf und f^q genau die Menge **aller** realisierbaren Köpfe an der Marke q. Um nun aus f^q entsprechende Intervallmengen zu extrahieren, wird ein Zusammenhang mit sog. Kofaktoren ausgenutzt. Dabei heißt eine Funktion $f_{x_i} := f(x_1, ...x_{i-1}, 1, x_{i+1}, ...x_n)$ **positiver Kofaktor** und $f_{x_i'} := f(x_1, ...x_{i-1}, 0, x_{i+1}, ...x_n)$ **negativer Kofaktor** von f. Man bezeichnet $f_{ab} := (f_a)_b$ als iterierten Kofaktor. Wir schreiben $f[a = \alpha]$ für den

[8] einschließlich Arrayvariablen mit entsprechender Vielfachheit und Verbunden

Kofaktor f_a, falls $\alpha = 1$ bzw. $f_{a'}$, falls $\alpha = 0$. Die **ON-Menge** von f ist eine Teilmenge des Definitionsbereichs $\{0,1\}^n$ von f und besteht aus den Elementen $\alpha \in \{0,1\}^n$, für die gilt $f(\alpha) = 1$. Weitere Details finden sich in [21, 22].

Lemma 2. ***(Exakte Wertbelegungsmenge einer Variable)***
Für die charakteristische Funktion der Menge aller ralisierbaren Wertebelegungen $f^{a_i[d],q} : \{0,1\}^{k_i} \rightarrow \{0,1\}$ *einer*[9] *Variable* $a_i[d]$ *an Marke* q*, gilt:* $f^{a_i[d],q}(x_1, x_2, \ldots x_{k_i}) = 1$ *gdw.* $f^q[x_{i,d,1} = x_1][x_{i,d,2} = x_2] \ldots [x_{i,d,k_i} = x_{k_i}] \neq \emptyset$.

Proof. Ergibt sich aus den Definitionen. □

Nun muss die Funktion $f^{a_i[d],q}$ gar nicht explizit berechnet werden, wenn die Intervallmengen nicht als z.B. ROBDD[10] benötigt werden. Insbesondere auch dann nicht, wenn lediglich eine untere und obere Schranke für Variablen zu bestimmen ist. Statt dessen genügt im Falle der Repräsentation von f^q als ROBDD die Prüfung von $f^q[x_{i,d,1} = x_1][x_{i,d,2} = x_2] \ldots [x_{i,d,k_i} = x_{k_i}]$ auf Leerheit (also bei ROBDDs der Test auf 0). Es muss lediglich die Intervallmenge (ON-Menge) von $f^{a_i[d],q}$ bestimmt werden, welche aber auch direkt aus f^q extrahiert werden kann. Ein zusätzlicher Schritt zur Bestimmung der Intervallmengen aus $f^{a_i[d],q}$ und auch die explizite Berechnung von $f^{a_i[d],q}$ ist damit nicht nötig.

Theorem 3. ***(Polynomielle Laufzeit exakter Wertebereichsanalysen)***
Die Bestimmung exakter Wertebereiche in Form von Intervallmengen eines Kellersystems $(P, \Delta, \hookrightarrow)$ *benötigt polynomielle Zeit in der Größe dieses Kellersystems.*

Proof. Die Berechnung des Post*-Automaten zu einem gegebenen Kellersystem $(P, \Delta, \hookrightarrow)$ benötigt die Zeit $O(|P||\Delta|(|\hookrightarrow| + |\Delta|))$ (siehe [23]). Der Post*-Automat enthält nach Konstruktion Transitionen von jedem Initialzustand s zur Marke q, welche sämtliche realisierbaren Pfade des Modells zusammenfassen. Zur Bestimmung der charakteristischen Funktionen f^q sind daher lediglich die Zielköpfe dieser Transitionen zu extrahieren, was für alle Marken zusammen insgesamt amortisiert O(Größe des Post*-Automaten als ROBDD) Zeit benötigt. Je Variable ist es dann im ROBDD von f^q nötig, die ON-Menge (repräsentiert als Intervallmenge) von $f^{a_i[d],q}$ zu bestimmen, was leicht modifiziert[11] nach [21, 22] polynomiell in der ROBDD-Größe von f^q geht. Also benötigt das Verfahren insgesamt eine Zeit: polynomiell in der Modellgröße. □

Sind lediglich untere und obere Schranken für SPDS-Variablen zu bestimmen, so kann auch die Bestimmung der ON-Menge von $f^{a_i[d],q}$ entfallen. In diesem Fall genügt ein einziger Tiefensuchlauf im ROBDD von f^q aus, um aus der ON-Menge von $f^{a_i[d],q}$ die kleinste und größte Variablenbelegung α und β zu bestimmen, so dass gilt ($\leq$ bezeichne dabei die lexikographische Ordnung):

$$f^{a_i[d],q}(\alpha) = f^{a_i[d],q}(\beta) = 1 \wedge \forall \zeta : ((f^{a_i[d],q}(\zeta) = 1) \Rightarrow \alpha \leq \zeta \leq \beta). \qquad (3)$$

[9] wie in Listing 1.4 definierten
[10] reduziert geordnetes binäres Entscheidungsdiagramm [21, 22]
[11] Die ON-Menge für einen Teil der Variablen des Definitionsbereichs kann ähnlich berechnet werden, wie die ON-Menge selbst.

5 Rückinterpretation in die Hochsprache

Wie in Listing 1.2 am Beispiel des übergebenen Parameters n zu sehen, korrelieren bei der Modellgenerierung mittels JMoped Modellvariablen direkt mit Programmvariablen. Lokale und Paramtervariablen werden bei JMoped konventionsgemäß in den Remopla-Modellen mit $v0, v1, \ldots$ bezeichnet, wobei zuerst für alle Paramtervariablen Namen vergeben werden. Die Reihenfolge der Definitionen entspricht der im Quellprogramm. Bei Anwendung von Optimierungen durch **HalSPSI** kann diese Ordnung allerdings verloren gehen. Mittels zusätzlicher Namensverlängerung bei der Modellgenerierung ist eine problemlose Zuordnung der exakten Wertebereiche aus den SPDS zu den Variablen des Quellprogramms möglich. Dies gilt aber wiederum auch nur dann, wenn diese Variablen nicht durch **HalSPSI** wegoptimiert wurden[12]. Zu den $v0, v1, \ldots$ kommen Hilfvariablen $s0, s1, \ldots$ hinzu, welche den lokalen Operanden-Stack einer Methode in Java simulieren. Da dieser Operanden-Stack a priori bereits im Bytecode in seiner Tiefe beschränkt ist, wird zur Simulation des Operanden-Stacks kein realer Stack des Kellersystems benötigt. Die Wertebereiche der Variablen $s0, s1, \ldots$ sind insofern also für eine Rückinterpretation uninteressant. Die Wertebereiche der restlichen Variablen sind aber nur dann auch exakt für das Quellprogramm, wenn das generierte Modell das Programmverhalten zu 100% beschreibt. Für das Beispiel in Listing 1.2 trifft dies nur zu, wenn n und der Rückgabewert der Funktion fac in Listing 1.1 stets nichtnegativ und kleiner als 65536 sind. Neben vielen weiteren Beispielen, können ganze Klassen oder große Teile von Programmiersprachen direkt durch SPDS ausgedrückt werden. So wurde in [1] eine Teilsprache von ISO C direkt als SPDS formuliert. Letztendlich muss es aber für turingmächtige Programmiersprachen stets Beispiele geben, welche durch die Modellierung als SPDS Präzision in ihrer Semantik verlieren. In diesen Fällen muss bei der Modellgenerierung auf den Erhalt der Konservativität geachtet werden. Auch JMoped muss z.B. dazu derartig in der Modellgenerierung angepasst werden, dass z.B. statt auftretender Speicherüberläufe im Modell auf Grund eines zu klein modellierten Heaps die SPDS-Pfade mit z.B. undefinierten Datenwerten fortgesetzt werden. So entstehen zwar größere exakte Intervallmengen für die Modelle, die dann aber als immer noch konservativ auch für das Quellprogramm verwendet werden können.

6 Experimente

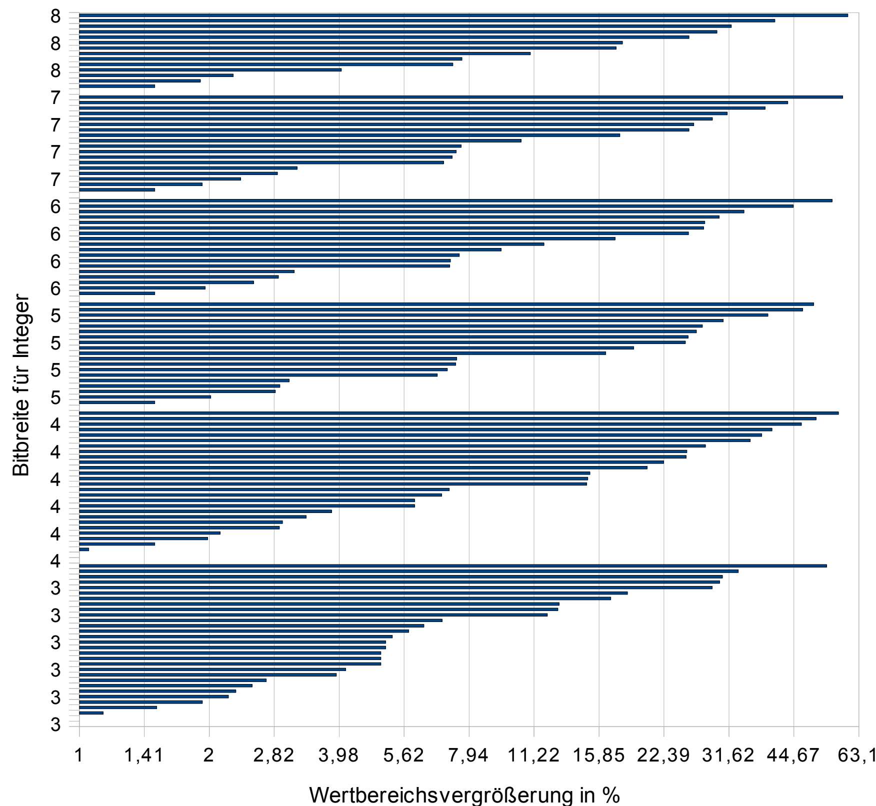
Als Grundlage für die Experimente dienten 191 von 240 Remopla-Modellen, zu denen exakte Wertebereiche berechnet werden konnten. Die Modelle wurden mittels JMoped aus Java Beispielen gewonnen, welche zum größten Teil zu den Benchmark-Instanzen der Werkzeuge JMoped bzw. Moped gehören. Zur Modellgenerierung wurden jeweils unterschiedliche Bitbreiten (bis zu 8 Bit) zur Modellierung von Ganzzahlen (Integer) verwendet. Aufgabe war es, zu diesen Modellen

[12] Änderungen durch Optimierungen sind dann rückverfolgbar zu halten.

einmal mit der konservativen und einmal mit der exakten Wertebereichsanalyse Intervallmengen zu bestimmen. Durchgeführt wurden die Experimente mit einem AMD 64 X2 4200+ mit 2 GB Hauptspeicher unter Linux (Kernel 2.6.27).

Die Laufzeiten der konservativen und exakten Wertebereichsanalyse unterscheiden sich um einige Größenordnungen. Die Berechnung der exakten Intervallmengen benötigte (wie auch theoretisch nachgewiesen in Theorem 2) oft mehr Zeit als eine Modellprüfung, während das konservative Verfahren sehr viel kleinere Laufzeiten stets im Sekundenbereich liefert und auch deutlich größere Modelle zu analysieren vermag.

Abbildung 3. Approximationsfehler des Wertebereichs der konservativen Analyse



Durch die konservative Wertebereichsanalyse werden in den Beispielen im Durchschnitt die Intervallmengen um 11.65% größer[13]. Diese Vergrößerung entspricht dem Approximationsfehler, welcher in Abbildung 3 für die einzelnen Modelle (ab 3 Bit) zu sehen ist. Desto größer die Bitbreite für Integer modelliert

[13] Im Durchschnitt beträgt die Präzision der konservativen Analyse also 88,35%.

wurde, desto seltener konnten wegen Speicherüberläufen exakte Intervallmengen bestimmt werden. In 6% der Fälle (vorranging kleine Bitbreiten) konnten die Intervallmengen bereits durch die konservative Wertebereichsanalyse exakt vorhergesagt werden (12 von 191 Fälle). Wie in Abbildung 3 auch zu erkennen, steigt bei Erhöhung der modellierten Bitbreite in einem Modell der Approximationsfehler nur leicht an. Dies läßt auf einen relativ stabil bleibenden Approximationsfehler bei größer werdenen Modellen schließen. Der Approximationsfehler ändert sich nicht signifikant, falls man lediglich die untere und obere Intervallgrenze (Minimal- und Maximalwert aus den Intervallmengen) zur Berechnung des Approximationsfehlers verwendet.

Insgesamt zeigt sich, dass mit rekursionspräzisen Analysen zur Bestimmung von Intervallmengen die Qualität des Analyseergebnises in vielen Fällen nur noch relativ wenig verbessert werden kann (man beachte die logarithmische Skala der X-Achse in Abbildung 3).

7 Verwandte Arbeiten

Ein Problem gängiger Intervallanalysen ist die Fehlabstraktion der Semantik der Ringarithmetik mittels unbeschränkter Datentypen, welches sich gerade bei kleineren Bitbreiten, wie sie bei der Modellprüfung eingesetzt werden, zum Tragen kommt. So missachten z.B. Intervallanalysen, wenn sie mittels linearen Programmen Variablen-Grenzen bestimmen, Ringarithmetik [8]. Sind aber im Modell unbeschränkte Ganzzahlen erlaubt, so kann für Fixpunktalgorithmen keine Konvergenz mehr garantiert werden [7]. Mittels Aufweitung (widening) und Einengung (narrowing) [16, 7] oder Beschränkung der Fixpunktiterationen [24] kann Konvergenz nur noch mit einhergehendem Qualitätsverlust erreicht werden. So erlangt man aber nur noch eine ungenaue sichere Überapproximation der Analyseergebnisse [25]. Die hier betrachteten SPDS verfügen über beschränkte Ganzzahlen, womit Konvergenz auch ohne Aufweitung und Einengung möglich ist. Nach [26] werden zudem z.B. Aufweitungs-Operatoren auch von Hand erstellt und müssen bei unzureichender Genauigkeit immer wieder neu angepasst werden. Daher verwendet **HalSPSI** einen Fixpunktansatz (wie auch Delzanno [6]) für die konservative Intervallanalyse mit präzisen Ergebnissen ohne Aufweitung bzw. Einengung, welche es zudem gegenüber von Ungleichungssystemen (z.B. Polyeder[14] [26]) ermöglicht, mehrere Intervalle je Variable gleichzeitig in Form von Intervallmengen zu behandeln und nicht nur eine untere und eine obere Schranke liefert [24]. Auch unterliegt der Ansatz keinen Monotonie-Einschränkungen wie in [25] oder Einschränkungen auf eine geringe Anzahl spezieller Operationen. Bodik, Gupta und Sarkar [27] berücksichtigen z.B. nur Zuweisung einer Konstante und Addition einer Konstante. Vielmehr wird die Ausdrucksauswertung hier ohne Einschränkungen der Operanden oder Operationen wie Multiplikation, Division und Modulo (Restbestimmung) interpretiert. Eingesetzt werden Intervallanalysen u.A. bei der Elimination überflüssiger Prüfungen von Feldzugriffen (Array Bound Checks) [24]. Cousot, Halbwachs und Schwarz (et al.) nutzen dazu

[14] engl. polyhedra

Abstrakte Interpretation mit statischer Datenfluss-Analyse [6], um überflüssige Prüfungen von Feldzugriffen zu identifizieren.

Insgesamt ist aber die maximal mögliche Bestimmung von Wertebereichen von Variablen in all diesen Publikationen nicht das Hauptziel. Unsere Ansätze sind zudem weitgehend unabhängig von der Quellsprache und können im Gegensatz zu Hochsprachen exakt durchgeführt werden [3]. Dies führt zu sehr präzisen Analysen für die Ausgangssprache, welche nicht auf endliche Modelle beschränkt sind und beliebige Rekursion berücksichtigen.

8 Zusammenfassung und Ausblick

Es wurde eine Methode vorgestellt, die in Experimenten durch Speicherbegrenzung in Java-Prgrammen sehr genaue Wertebereiche der Variablen bestimmt. Die Java-Programme wurden dazu zunächst in ein Rekursionsmodell überführt, welches präzise das rekursive Verhalten des Programms beschreibt. Für diese Rekursionsmodelle konnten exakte Wertebereiche berechnet werden. Je nach Art der Modellgenerierung lassen diese Wertebereiche im Modell Rückschlüsse auf die Wertebereiche des ursprünglichen Java-Programms zu. Da die Bestimmung exakter Wertebereiche (nicht nur komplexitätstheoretisch) aufwändig ist, sind möglichst kleine Modelle für die Durchführbarkeit entscheidend. Die Modellgenerierung mittels JMoped ermöglicht die Modellierung ganzzahliger Datentypen mit nur wenigen Bits und beschränkt somit automatisch den adressierbaren Speicher, in welchem sich die zur Laufzeit erzeugten Objekte befinden, was bereits in 80% (191 von 240) der untersuchten Fälle genügte. Eine Modell-Verkleinerung bzw. -Verbesserung führt in diesen Fällen dann nur noch zu einer Beschleunigung der rekursionspräzisen Intervallanalyse. Wie auch in den Experimenten zu erkennen war, können mit konservativen Analysen, welche um Größenordnungen schneller sind als exakte Verfahren[15], bereits 88% Präzision erreicht werden. Dennoch können nicht nur approximative, sondern auch exakte Methoden genutzt werden, um präziser das Verhalten und Eigenschaften von SPDS und damit von C, Java oder anderen Programmen vorherzusagen. Zur Beschleunigung der exakten Wertebereichsanalyse können zusätzlich Reduktionsverfahren unseres Werkzeugs **HalSPSI** eingesetzt werden, wie z.B. die Wertebereichsreduktion, Stotterreduktion oder Slicing [2–5]. So können dann noch größere Modelle analysiert und Intervallmengen der Modelle schneller berechnet werden.

Genauer zu betrachten bleibt eine bezüglich Wertebereichsanalyse nachweislich konservative Modellgenerierung aus höheren Programmiersprachen (und nicht nur Java wie in Abschnitt 5). Für z.B. eingebette Systeme, welche u.U. nur einen Teil von ISO C verwenden (wie in [1] erläutert), degeneriert eine solche Modellgenerierung im Idealfall zur Identität. Ein Nachweis zum Erhalt von Konservativität kann dann entfallen.

[15] Obwohl die vorgestellte exakte Wertebereichsanalyse nur polynomielle Laufzeit benötigt.

Literatur

1. R. Kirner, W. Zimmermann, D. Richter: On Undecidability Results of Real Programming Languages. Im Rahmen des 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS) (2009)
2. D. Richter, W. Zimmermann: Slicing zur Modellreduktion von symbolischen Kellersystemen. Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel (2007)
3. D. Richter: Modellreduktionstechniken für symbolische Kellersysteme. Proc. of the 25. Workshop 'Programmiersprachen und Rechenkonzepte', University Kiel (2008)
4. D. Richter, W. Zimmermann: Variablenelimination für symbolische Modelle. Im Rahmen der 39. GI-Jahrestagung zum 4. Workshop 'Modellbasiertes Testen', Lübeck (2009)
5. D. Richter: Äquivalenzanalysen - exakt oder nicht - im Vergleich. Im Rahmen des 26. Workshops 'Programmiersprachen und Rechenkonzepte', University Kiel (2009)
6. Thi Viet Nga Nguyen, Francois Irigoin: Interprocedural program analyses for efficient array bound checking. In: 2nd International Workshop on Automated Program Analysis, Testing and Verification (WAPATV), ACM Press New York (2001) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.1279.
7. Patrick Cousot, Radhia Cousot: Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In: Programming Language Implementation and Logic Programming. Volume 631 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag Berlin Heidelberg (1992) 269–295 http://www.springerlink.com/content/p869x76457527706/.
8. Jeffery Von Ronne, Kleanthis Psarris, David Niedzielski: Verifiable Range Analysis Annotations for Array Bounds Check Elimination. 13th International Workshop on Compilers for Parallel Computers (CPC). Lisbon, Portugal. (2007)
9. S. Muchnick: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
10. S. Schwoon: Model-Checking Pushdown Systems. Technische Universität München (2002)
11. S. Kiefer, S. Schwoon, D. Suwimonteerabuth: Introduction to Remopla. Institute of Formal Methods in Computer Science, University of Stuttgart (2006)
12. J. Esparza, S. Schwoon: A BDD-based model checker for recursive programs. LNCS Volume 2102, 324-336, Springer (2001)
13. J. Obdrzalek: Formal verification of sequential systems with infinitely many states. Master's Thesis, FI MU Brno, Masaryk University (2001)
14. J. Obdrzalek. In: Model Checking Java Using Pushdown Systems. LFCS, University of Edinburgh (2002)
15. William H. Harrison: Compiler Analysis of the Value Ranges for Variables. In IEEE Transactions on Software Engineering, Vol 3, Issue 3, 243-250 (1977)
16. Patrick Cousot, Nicolas Halbwachs: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press New York (1978) 84–96 http://portal.acm.org/citation.cfm?id=512770.
17. Thomas Gawlitza, Jan Reineke, Helmut Seidl, Reinhard Wilhelm: Polynomial Precise Interval Analysis Revisited. Technical Report, TU Muenchen (2006)
18. J. Whaley, M.S.L.: Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. Proc. of the SIGPLAN 2004 conference on Programming language design and implementation, 131-144, ACM (2004)

19. Florian Martin: Experimental comparison of call string and functional approaches to interprocedural analysis. In: Compiler Construction. Volume 1575., Lecture Notes in Computer Science (LNCS) (1999) http://www.springerlink.com/content/u5xkkgvf40tnaq9u/.
20. Bageshri Karkare, Uday P. Khedker: An improved bound for call strings based interprocedural analysis of bit vector frameworks. In: ACM Transactions on Programming Languages and Systems (TOPLAS). Volume 29(6)., ACM Press New York (2007) http://portal.acm.org/citation.cfm?id=1286829.
21. P. Molitor, C. Scholl: Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen. Teubner Verlag (1999)
22. P. Molitor, J. Mohnke: Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods. Springer Netherlands (2004)
23. J. Esparza, D. Hansel, P. Rossmanith, S. Schwoon: Efficient algorithms for model checking pushdown systems. Proc. of the 12th International Conference on Computer Aided Verification, LNCS 1855 (2000)
24. Radu Rugina, Martin C. Rinard: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In: ACM Transactions on Programming Languages and Systems (TOPLAS). Volume 27(2)., ACM Press New York (2005) 185–235 http://portal.acm.org/citation.cfm?id=1057388.
25. Thomas Gawlitza, Jan Reineke, Helmut Seidl, Reinhard Wilhelm: Polynomial precise interval analysis revisited. Technical Report, TU München, Germany (2006) http://rw4.cs.uni-sb.de/publications.shtml.
26. Chao Wang, Zijiang Yang, Aarti Gupta, Franjo Ivancic: Using counterexamples for improving the precision of reachability computation with polyhedra. In: Proceedings of the 19th International Conference on Computer Aided Verification. Volume 4590 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag Berlin Heidelberg (2007) 352–365 http://www.springerlink.com/content/83835rn353287342/.
27. Rastislav Bodik, Rajiv Gupta, Vivek Sarkar: Abcd: eliminating array bounds checks on demand. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, ACM Press New York (2000) 321–333 http://portal.acm.org/citation.cfm?id=349342.

A New Look on Data Parallelism: Space vs. Time

Sven-Bodo Scholz
University of Hertfordshire

A New Look on Data Parallelism: Space vs. Time

Data Parallelism often is used synonymously with "forallloops or "mapfunctions. This leads to a perception of Data Parallelism as being a niche approach, mainly suitable for autoparallelisation attempts in the context of scientific codes written in Fortran or similar languages. In this talk we try to present a new look at Data Paralellism which suggests that Data Parallelism as programming model has far wider applicability. Furthermore, it seems that it is inherently better suited than many other approaches when it comes to generating code for various different modern multicore architectures.

Statische Erkennung von semantischen Feature-Interaktionen

Wolfgang Scholz
Universität Passau

Interaktionen zwischen verschiedenen Programmteilen erschweren die Entwicklung komplexer Softwaresysteme. Feature-orientierte Softwareentwicklung hat das Ziel, Programmfeatures in verschiedenen Kombinationen zu einem korrekten Programm zusammenzufügen, und bietet damit aussichtsreiche Möglichkeiten für die modulare Programmentwicklung. Die Erkennung und Behebung von ungewollten Feature-Interaktionen wird bisher manuell vorgenommen und ist ein aufwändiger Prozess, da er detailliertes Wissen über die Implementierung aller involvierten Features erfordert. Um die Effektivität von Feature-orientierter Programmierung zu erhöhen ist es daher notwendig, Werkzeuge zur Verfügung zu stellen, die in der Lage sind, semantische Feature-Interaktionen aufzudecken.

Ziel des Projekts ist es, auf der Basis des Eclipse-Plugins *FeatureIDE* [1] und der Verifikationsplattform *Why* [2] ein Programmierwerkzeug zu entwickeln, mit dem es möglich ist, eine bestimmte Klasse von semantischen Feature-Interaktionen in Java-Software-Produktlinien statisch bei der Kompilation zu erkennen.

Literatur

1. C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, S. Apel. *FeatureIDE: Tool Framework for Feature-Oriented Software Development*, Proceedings of the 31th International Conference on Software Engineering (ICSE), pages 611–614. IEEE Computer Society, May 2009.
2. J. Filliâtre *Verification of Non-Functional Programs using Interpretations in Type Theory*, Journal of Functional Programming, 13 (4), pages 709-745, 2003.

ARAL: a Language for Information Exchange between Program Analysis Tools

Markus Schordan
Fachhochschule Technikum Wien

The analysis results annotation language (ARAL) is designed to be a general format that allows the exchange of analysis information between program analysis tools either in a separate file or through program annotations. It is suitable for representing flow-sensitive and context-sensitive information and aims at supporting a wide range of analyses with focus on data-flow analysis and abstract interpretation in general. For example, the language is general enough to represent any analysis information that is computed by AbsInt's Program Analyzer Generator (PAG). Beside exchange of analysis information, the purpose of the language also is the support of testing analyzers and allowing to manually add annotations, as is important for worst-case execution time analysis. The development of ARAL is motivated by the need of making analysis results persistent for enabling whole-program analysis of very-large software, and the goal of information exchange between program analysis tools in the ALL-TIMES project.

Robots, Software, Mayhem?
Towards a Design Methodology for Robotic Software Systems

Dietmar Schreiner
Technische Universität Wien

The development of autonomous robotic systems has experienced a remarkable boost within the last years. Away from stationary manufacturing units, current robotic systems have grown up into autonomous, mobile systems that not only interact with real world environments, but also fulfill mission critical tasks in collaboration with human individuals on a reliable basis. Therefore, todays robotic systems can be described as highly reactive, inherent parallel, distributed real-time systems. Our work aims at an improved software development methodology, that on the one hand allows high level development of certifiable robotic software, but on the other hand is capable of synthesizing optimized low-level code for robust concurrent real-time environments.

A Formalisation of the OSEK Concurrency Model

Martin Schwarz
Technische Universität München

The OSEK specification was created to increase portability of applications written for embedded systems with interrupt-driven concurrency. Despite of the complex control-flow due to scheduling and interrupts, programs to be executed on an OSEK-compliant system are meant to exhibit an essentially sequential behaviour. Our goal is to make this explicit and exploit it for transferring techniques for the analysis of sequential programs to OSEK programs. Building on that, we define a notion of races for interrupt-driven programs and provide a precise algorithm for detecting all races.

The OSEK specification defines a unified operating system (OSEK OS), which executes the tasks and interrupts of OSEK programs in a well-defined manner and provides a set of library functions for resource management and scheduling. The basic scheduling policy of OSEK OS is to work through the activated tasks in priority order. Once started, a task will run to termination unless a task of higher priority is activated and pre-empts it, i.e. no time-slicing is used. This property allows the translation of OSEK programs to a sequential, stack-based execution model.

For synchronisation the Priority Ceiling Protocol (PCP) is used. The key idea of the PCP is to raise the priority of a task which has acquired a resource to the highest priority of all tasks declared to use that resource. This policy ensures dead-lock freedom and minimises priority inversion, where a high-priority task waits for a lower-priority task to release a resource.

Lazy Continuations for Java Virtual Machines

Lukas Stadler
Johannes Kepler Universität Linz

Continuations, or “the rest of the computation”, are a concept that is most often used in the context of functional and dynamic programming languages. Implementations of such languages that work on top of the Java virtual machine (JVM) have traditionally been complicated by the lack of continuations because they must be simulated.

This talk will present our implementation of continuations in the Java virtual machine with a lazy or on-demand approach. Our system imposes zero run-time overhead as long as no activations need to be saved and restored and performs well when continuations are used. Although our implementation can be used from Java code directly, it is mainly intended to facilitate the creation of frameworks that allow other functional or dynamic languages to be executed on a Java virtual machine.

Along with some preliminary performance numbers this talk will also feature a discussion on how the concept of continuations fits into the Java world and which problems and needs it addresses.

Communication-based Development of Systems Using Standard Programming Languages (Extended Abstract)

Annette Stümpel

Institute of Software Technology and Programming Languages
University of Lübeck, Lübeck, Germany
stuempel@isp.uni-luebeck.de
http://www.isp.uni-luebeck.de

1 Proposal

Implementations often do not show much resemblance to the initial specifications describing the communication between the system components. In order to reduce this discrepancy and to strengthen the impact of communication-based specifications, we promote a communication-oriented style of programming, which is possible in some standard programming languages.

2 Motivation

Modern software systems are usually structured into communicating subsystems and components, see Fig. 1. A user of the system can often observe only the communication between the components and the environment. The internal design, such as the state and the algorithms inside the components, remains hidden.

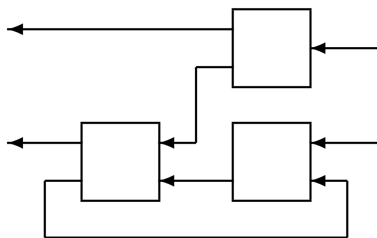


Fig. 1. System composed of communicating components

A specification of a system first describes the black box view, which concentrates on the observable behaviour. The communication of the system components with each other and with their environment characterizes this external

view. Modelling whole conversations is also essential for the specification of (web) services ([1, 4]).

For the communication-based specification software developers employ graphical models such as message sequence charts [6] or UML sequence diagrams [7] or even less formal notations. Stream processing [2, 9] constitutes a formal model for communication-based specifications. Stream functions describe the input / output behaviour of components and can be composed to larger systems.

Although the communication-based view is the essential part of the specification which the programmer as well as the user understand, the implementation often does not reflect that specification. There is a gap between the communication-based specification and the implementation because the implementation is often not constructed from the communication-based specification.

With some programming languages, even some standard ones, it is possible to mould the communication-based specification into a specification which clearly reflects the underlying communication-based specification. We show how to achieve this with a lazy functional language, Haskell, and an object oriented language with threads, Java.

The communication-oriented implementations can be obtained from communication-based specifications in the setting of stream processing. In an initial step, these input / output specifications are enriched with states by a design decision and formal transformations [3]. The resulting state transition machine with input and output establishes the relationship between the previous input history of a component and its current state. Depending on the current state and the current input message the state transition machine records the outputs caused by the input and the successor state.

3 Haskell

The functional language Haskell supports communication-based programming with its lazy lists.

For a straightforward implementation of a network of components in Haskell, each component is implemented by a function operating on lists. We introduce a name for each communication channel.

We obtain the composite network as mutually recursive equations: For each component an equation determines the tuple of output streams referred to by their names resulting from applying the component's function to the tuple of input streams also referred to by their names.

Lazy lists are well suited for modelling the operational progress of the system with lists representing the contents of the communication channels. A lazy list is a list whose elements are already evaluated in an initial part. At a certain point the remaining list is unknown. It is even unknown whether there are further elements in the list at all. This concept corresponds exactly to a stream, which records the sequence of messages transmitted on a channel until a certain point of time: the initial part has already been observed but it is not clear whether further messages will appear, let alone their content.

For example, the empty stream $\langle\rangle$ corresponds to the empty lazy list $\perp$. The stream $\langle 2, 6, 3, 1\rangle$ corresponds to the lazy list `2 : 6 : 3 : 1 :` $\perp$. Note that there is no direct syntax for lazy lists in the programming language Haskell.

The laziness of the stream lists has to be taken carefully into account for the implementation of the components by functions over lists. Simple transformations on specifications may not transfer from the stream notation to the Haskell notation. Haskell functions which refer to larger parts of the input stream may block the whole network. Haskell functions which refer only to the first element of their input streams do not block the network. A state transition table of a state transition machine validates this shape of specification.

If the components' communication-based specifications are already provided in the form of a state transition machine, the network can straightforwardly be implemented in Haskell using lazy lists.

4 Java

Java provides threads for communication-based programming. A straightforward approach implements the components and the channels as concurrent threads. Each channel thread records the messages sent via the channel and not yet read by the recipient in a FiFo-queue, and each component thread continually reads messages from its connected input channels and writes messages to its connected output channels.

The STREAMS tool developed at the University of Lübeck [5] supports the simulation of systems in Java using this approach. The STREAMS simulator is a graphical tool which visualizes the stepwise execution of systems. The tool provides an editor for building systems from the implemented components and connecting output ports of components with input ports with the same type. Furthermore, the tool provides a simulation mode in which the user can manually select the component for the next execution step or the user can select between various concurrent simulation modes in which the components can execute several steps concurrently.

The tool provides implementations for channels and superclasses for the components. The state transition machines can be coded in a quite straightforward way. Each component must contain a method implementing the single transition steps. This method may use the following important methods for accessing its input and output channels:

- checking whether an input channel is empty
- inspecting the first element of an input channel
- removing the first element of an input channel
- writing an element to an output channel

Each component is equipped with a standard graphical representation. Enhanced graphical representations can be obtained by overwriting some simple methods.

In this way we get implementations in Java which clearly reflect the communication-based specifications. In large parts these implementations can even be

generated automatically from state transition tables of state transition machines [8].

5 Conclusion

Communication-based specifications play an essential rôle in the development of systems built from communicating components. Such systems can only operate successfully if each component always provides exactly the desired service. Programs which are systematically constructed from these specifications are supposed to be developed more efficiently and to deserve more confidence in their correctness than programs which do not reflect the specified communication.

Besides languages which are designed for the communication-based implementation, such as Erlang, there are also some standard programming languages which can be used for a communication-oriented programming style. Haskell with its lazy lists and Java with its threads are good examples.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer, 2004.
2. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Monographs in Computer Science. Springer, 2001.
3. W. Dosch and A. Stümpel. Deriving state–based implementations of interactive components with history abstractions. In I. Virbitskaite and A. Voronkov, editors, *Perspectives of Systems Informatics (PSI 2006)*, number 4378 in Lecture Notes in Computer Science, pages 180–194. Springer, 2007.
4. W. Dosch and A. Stümpel. Implementing services by partial state machines. In M. Nielson, A. Kučera, P. Miltersen, C. Palamidessi, P. Tůma, and F. Valencia, editors, *Theory and Practice of Computer Science (SOFSEM 2009)*, number 5404 in Lecture Notes in Computer Science, pages 241–254. Springer, 2009.
5. B. Hoffmeister. Entwicklung eines Simulators zur Darstellung der Kommunikation in verteilten Systemen. Semesterarbeit am Institut für Softwaretechnik und Programmiersprachen der Medizinischen Universität zu Lübeck, 2001.
6. International Telecommunication Union (ITU-T). *Z.120: Message Sequence Chart (MSC)*, 2004.
7. Object Management Group (OMG). *Unified Modeling Language: Superstructure*, 2.2 edition, 2009.
8. A. Stackebrandt. Component-based development of ePayment systems. Diplomarbeit, Universität zu Lübeck, 2008.
9. A. Stümpel. *Stream Based Design of Distributed Systems through Refinement.* Logos Verlag Berlin, 2003.

FQL: A Query Language for Program Testing *

Andreas Holzer Christian Schallhart Michael Tautschnig Helmut Veith

Formal Methods in Systems Engineering, FB Informatik, TU Darmstadt
holzer@forsyte.de, schallhart@forsyte.de, tautschnig@forsyte.de, veith@forsyte.de

Abstract. We present a new approach to program testing which enables the programmer to specify test suites in terms of a versatile query language. Our query language subsumes standard coverage criteria ranging from simple basic block coverage all the way to predicate complete coverage and multiple condition coverage, but also facilitates on-the-fly requests for test suites specific to the code structure, to external requirements, or to ad hoc needs arising in program understanding/exploration. The query language is supported by a model checking backend which employs the CBMC framework. Our main algorithmic contribution is a method called *iterative constraint strengthening* which enables us to solve a query for an arbitrary coverage criterion by a single call to the model checker and a novel form of incremental SAT solving: Whenever the SAT solver finds a solution, our algorithm compares this solution against the coverage criterion, and strengthens the clause database with additional clauses which exclude redundant new solutions. We demonstrate the scalability of our approach and its ability to compute compact test suites with experiments involving device drivers, automotive controllers, and open source projects.

1 Introduction

In industrial software engineering, program testing is to remain the pivotal debugging and validation technology. While randomized and directed testing are important to achieve global assurance about software quality, and model-based testing helps to verify the conformance of the program with a high-level specification, there is practical need for a source code oriented white box testing methodology which assists the programmer in the software engineering cycle. Such a methodology should provide seamless support for code-driven testing, i.e., exploration of code under development, and requirement-driven testing for systematic quality assertion.

To address this need, we introduce a query language which combines easy navigation in real life C code with the ability to formulate complex coverage criteria. Providing straightforward queries for standard coverage criteria, our language FQL (FSHELL query language) aims to strike the right balance between expressiveness and simplicity. In FQL, a *program query* asks for a test suite following the general form

> **in** *prefix* **cover** *goals* **passing** *scope*

where the optional *prefix* directs the query to a specific program part, e.g., a source file or a function, *goals* describes the coverage criterion to be fulfilled by the test suite, and

* Published at VMCAI 2009 (Query-Driven Program Testing). Supported by DFG grant FORTAS – Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1).

an optional *scope* restricts test cases to pass through certain program paths only. For example, the query

```
> in /bla.c/cmp/ cover @blocks passing @call(err)
```

calls for a test suite which covers all blocks in function `cmp` of file `bla.c`, such that in all test cases, a call to `err` inside `cmp` is performed[1]. Other important and classical coverage criteria such as *predicate coverage*, *condition coverage*, *decision coverage*, *modified condition/decision coverage*, *predicate complete coverage* [1] etc. can also be expressed by natural queries in our language.

The added value of our language lies in the ability to define the query scope and the query goals in a quite flexible manner. Suppose for instance that in Listing 1 we want to cover (1) all calls to `cmp` and (2) all decisions inside `cmp`. This amounts to a coverage criterion which combines basic block coverage for (1) and decision coverage for (2). There are two different possibilities for this combination: either we want to cover the union of all call positions and all decisions, and write the query

```
> in /bla.c/ cover @call(cmp), cmp/@decisions
```

or we want to cover all possible *combinations* of calls to `cmp` and subsequent decisions inside `cmp`, i.e., the Cartesian product of the individual test goals:

```
> in /bla.c/ cover @call(cmp) -> cmp/@decisions
```

To ensure that, for each call site, each of the decisions is reached before the body of `cmp` is left, we write

```
> in /bla.c/ cover @call(cmp) -[@func(cmp)\@exit]> cmp/@decisions
```

Solutions to these queries are test suites. In case of Listing 1, these can be most easily described as sets of triples with input values for the variables x,y,z. For the first query, the singleton test suite $\{(1,0,1)\}$ covers all blocks and decisions in Listing 1; for the second and third case, possible solutions are $\{(1,0,1),(2,0,1),(1,0,2)\}$ and $\{(1,0,0),(0,1,0),(0,0,1)\}$ respectively. Note that the first test suite does not satisfy the second and third queries and the second suite does not satisfy the last query; the third solution, however, satisfies all three queries.

In Section 2 we build the mathematical foundation to formulate testing requirements as queries. In particular, we show how to state a query as a pair $\langle A,Q\rangle$ of automata over predicates. In this pair, the *observation automaton* A corresponds to the *scope* to be explored, and the *test goal automaton* Q specifies the *goals* to be covered. In Section 3, we provide an overview of our query language FQL and show how to translate FQL queries into such a pair $\langle A,Q\rangle$. Thus, the language reduces to a simple mathematical core in which we are able to formulate all relevant coverage criteria in a uniform way.

The second major contribution of this paper is an efficient query engine which integrates our theoretical framework, code instrumentation, bounded model checking, and SAT enumeration into a tool of high efficiency. Our query engine employs and adopts the software model checking framework of Kroening's CBMC [2]. Given a query $\langle A,Q\rangle$, our tool performs the following conceptual steps, cf. Figure 1:

[1] Expressions starting with "@", such as `@blocks`, typically denote sets of program locations, see Section 3.1 for a detailed explanation.

Listing 1. C source code of example `bla.c` with program counters

```
1  int cmp(int x, int y) {
2      int ¹value = 0;
3      if ²(x > y) ³value = 1;
4      else if ⁴(x < y) ⁵value = −1;
5      ⁶return value;
6  }

8  int main(int argc, char* argv[]) {
9      int x, y, z, xy, yz, xz;
10     ⁸x = ⁷input();
11     ¹⁰y = ⁹input();
12     ¹²z = ¹¹input();
13     ¹⁴xy = ¹³cmp(x, y);
14     ¹⁶yz = ¹⁵cmp(y, z);
15     ¹⁸xz = ¹⁷cmp(x, z);
16     if ²²(¹⁹xy == 1 && ²⁰yz == 1
17         && ²¹xz != 1)
18 ERROR: ²³err();
19     ²⁴return 0;
20 }
```

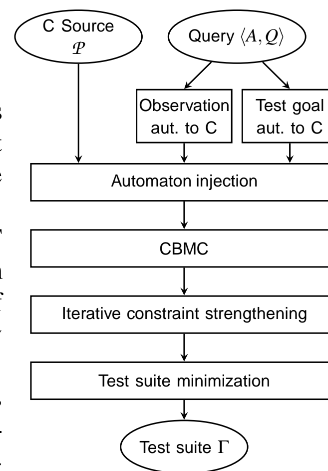


Fig. 1. Query processing

(1) We instrument the source code with monitors derived from the observation automaton A and the test goal automaton Q in such a way that the states of the monitors reflects the automata states.

(2) We use the code base of CBMC to obtain a SAT instance ϕ whose solutions correspond to the program paths π in the scope given by A. The instrumentation of step (1) guarantees that for each solution, we can easily determine which goals of Q are covered.

(3) We use the SAT solver to enumerate test cases as solutions to the SAT instance until we satisfy the coverage criterion defined by the query. The *iterative constraint strengthening* technique (ICS) used in this step is discussed below.

(4) To remove redundant test cases, we perform a test suite minimization. In our current implementation, we only do a simple post-processing; in future work, we plan to implement more aggressive minimization strategies. Note that the ICS enumeration in (3) involves nondeterministic choices which may give leverage to accelerate the algorithm with suitable heuristics.

Iterative Constraint Strengthening. A naive implementation of step (3) above would either use SAT enumeration to compute an enormous number of test cases until the test goals are reached, or it would call the SAT solver for each query goal anew. In *iterative constraint strengthening (ICS)*, we circumvent both problems by modifying the clause database of the SAT solver on-the-fly. Whenever the SAT solver halts to output a solution, we compare the test case obtained from this solution against the test goals. Then we add new clauses to the clause database in such a way that the next solution is guaranteed to satisfy at least one hitherto uncovered test goal. In this way, we exploit incremental SAT solving to quickly enumerate a test suite of high quality: Since we only add new clauses to the clause database, the SAT solver is able to reuse information learned in prior invocations. A similar strategy is used in *groupwise constraint strengthening (GCS)*, a further refinement of ICS. In GCS, we address coverage criteria such as multiple condition coverage or predicate complete coverage which have a nominally

exponential number of test goals by partitioning these goals into a small number of groups characterized by a common compound goal.

We show that FSHELL has better practical performance than BLAST's test case generation facility [3]: On comparable hardware, our test suites are computed faster, and contain fewer test cases. Due to the minimization step, our results also improve on those reported in our previous tool paper [4].

Note that our choice of CBMC and bounded model checking as a query solving backend has advantages which come at a price: On the one hand, we achieve excellent performance and have the guarantee that the model-checker respects ANSI-C, which is important for low level code, our primary application area. On the other hand, a bounded model checking approach may be *unable to compute certain test cases* involving paths larger than the constant bound. It is easy to come up with examples where this situation will happen, but it is is detectable by CBMC and accounted for in our implementation; it has neither occurred in the experiments we did for comparison with BLAST, nor in our experiments based on real-life controller code. In future work, we plan to complement the CBMC backend with abstraction-based and randomized test case generation backends.

Related Work. Beyer et al. [3] use the C model checker BLAST [5] for test case generation, focusing on basic block coverage only. BLAST has a two level specification language [6]. On a low level they specify trace properties by observer automata written in a C-like manner. On a high level they relate these automata by reachability queries. In contrast to FSHELL, their language is tailored towards verification. Furthermore, BLAST is based on predicate abstraction whereas CBMC is a SAT-based bounded model checker. As our experiments show, we outperform BLAST regarding test case generation. Lee et al. [7,8] investigate test case generation with model checkers giving coverage criteria in temporal logics. Java PathFinder [9] and SAL2 [10] use model checkers for test case generation, but they do not support C semantics.

2 A Formal Testing Framework

Given a program $\mathcal{P}$, we consider the possibly infinite *transition system* $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$ induced by $\mathcal{P}$ which consists of the state space $\mathcal{S}$, a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and a non-empty set of initial states $I \subseteq \mathcal{S}$.

Definition 1 (State Sequences and Paths). *Given a transition system* $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$*, a* state sequence *is a finite and non-empty word* $\pi = \langle s_1, \ldots, s_n \rangle \in \mathcal{S}^+$ *of states* $s_i \in \mathcal{S}$*. The sequence* π *is a* path*, if* $\langle s_i, s_{i+1} \rangle \in \mathcal{R}$ *holds for all* $1 \leq i < n$ *and if* $s_1 \in I$*. For a state* $s \in \mathcal{S}$*, we write* $s \in \pi$*, iff* $s = s_i$ *holds for some* $1 \leq i \leq n$*, and we denote with* $\Pi^{\mathcal{T}} \subseteq \mathcal{S}^+$ *the* set of paths *of* $\mathcal{T}$.

We use *state predicates* to describe properties of individual program states and we use *path* and *path set predicates* in the description of individual test goals and coverage criteria.

Definition 2 (State, Path, & Path Set Predicates). *Given a transition system* $\mathcal{T} = \langle \mathcal{S}, \mathcal{R}, I \rangle$*, we define a* state predicate p *as a predicate on the state space* $\mathcal{S}$*, a* path

predicate ϕ *as a predicate over the set* $\Pi^{\mathcal{T}}$*, and a* path set predicate Φ *as a predicate over the sets of paths* $2^{\Pi^{\mathcal{T}}}$*. We write* $s \models p$ *iff a state* $s \in \mathcal{S}$ *satisfies* p*,* $\pi \models \phi$ *iff a path* $\pi \in \Pi^{\mathcal{T}}$ *satisfies* ϕ*, and* $\Gamma \models \Phi$ *iff a path set* $\Gamma \subseteq \Pi^{\mathcal{T}}$ *satisfies* Φ*.*

We call a state predicate p, a path predicate ϕ, or a path set predicate Φ *feasible over* $\mathcal{T}$*,* iff, respectively, there exists a state $s \in \mathcal{S}$ with $s \models p$, there exists a path $\pi \in \Pi^{\mathcal{T}}$ with $\pi \models \phi$, and there exists a path set $\Gamma \subseteq \Pi^{\mathcal{T}}$ with $\Gamma \models \Phi$. Frequently, we are looking for a path (path set) which *contains* a state (a path) which satisfies a given state (path) predicate—leading to an *implicit existential quantification:*

Definition 3 (Implicit Existential Quantification). *To evaluate a state predicate* p *over a path* π*, we implicitly interpret* p *to be existentially quantified, i.e.,* $\pi \models p$ *stands for* $\exists s \in \pi . s \models p$*. Analogously, a path predicate* ϕ *is existentially evaluated over a path set* Γ*, i.e.,* $\Gamma \models \phi$ *iff* $\exists \pi \in \Gamma . \pi \models \phi$*.*

Remark 1. Note that a path π can satisfy a state predicate p *and* its negation $\neg p$, if there exist two states $s, s' \in \pi$ with $s \models p$ and $s' \models \neg p$. Moreover, a state predicate p can also be interpreted over a path set Γ in the natural way, i.e., $\Gamma \models p$ iff $\exists \pi \in \Gamma . \exists s \in \pi . s \models p$.

Program Observations. We use sequences of state predicates (*traces*) to specify program paths. A trace matches a state sequence if each state in the sequence satisfies the corresponding predicate. A trace automaton is an automaton accepting traces; each trace in turn specifies a set of program paths.

Definition 4 (Traces and Trace Automata). *Let* P *be a finite set of state predicates and* $\mathcal{S}$ *be a state space. Then a* trace *is a finite non-empty word* $t = \langle t_1, \ldots, t_n \rangle \in P^+$*. A trace* matches *a state sequence* $\pi = \langle s_1, \ldots, s_n \rangle \in \mathcal{S}^+$ *(denoted with* $\pi \models t$*), iff* $s_i \models t_i$ *for all* $1 \leq i \leq n$*.*

A trace automaton *over* P *is a nondeterministic finite state automaton* A *accepting traces over the alphabet* P*. We write* $\mathcal{L}(A)$ *to denote the set of traces accepted by* A *and* $\mathsf{acc}(A)$ *to denote the set of accepting states of* A*. A trace automaton* A *over* P matches *a state sequence* π *(denoted with* $\pi \models A$*), iff there exists a trace* $t \in \mathcal{L}(A)$ *with* $\pi \models t$*.*

Remark 2. Although we have – for the sake of simplicity – defined trace automata as finite state automata, our framework naturally extends to other types of automata such as push-down automata for which we can construct C monitors, cf. Section 4.1.

We will use traces and trace automata as a natural tool for defining path predicates in the language FQL. In particular, we will employ trace automata for two distinct ends: First, as *observation automata* which *restrict the paths in* $\mathcal{T}$ to those required in a query; and second, as *test goal automata* which specify the individual test goals of a coverage criterion.

Definition 5 (Path Restriction by Observation Automata). *Let* $\mathcal{T}$ *be a transition system and* A *a trace automaton. Then we define the set of paths in* $\mathcal{T}$ *restricted by observation automaton* A *as* $\Pi_A^{\mathcal{T}} = \{\pi \in \Pi^{\mathcal{T}} \mid \pi \models A\}$*.*

Coverage Criteria. In the framework of this paper, we define a *test case* to be a single path in $\Pi_A^{\mathcal{T}}$ and a *test suite* as a subset of $\Pi_A^{\mathcal{T}}$. Correspondingly, a *coverage criterion* imposes a predicate on test suites:

Definition 6 (Test Case & Test Suite). *Let $\mathcal{T}$ be a transition system and let A be an observation automaton for $\mathcal{T}$. Then a* test case *for the set of paths $\Pi_A^{\mathcal{T}}$ is a single path $\pi \in \Pi_A^{\mathcal{T}}$ and a* test suite *Γ is a finite subset $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ of the paths in $\Pi_A^{\mathcal{T}}$.*

Definition 7 (Coverage Criterion). *A* coverage criterion *Φ is a mapping from a transition system $\mathcal{T}$ and an observation automaton A to a path set predicate $\Phi_A^{\mathcal{T}}$ over $2^{\Pi^{\mathcal{T}}}$. We say that $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ satisfies the coverage criterion Φ on $\mathcal{T}$ under the restriction A iff $\Gamma \models \Phi_A^{\mathcal{T}}$ holds.*

While our definition of coverage criteria is very general, most coverage criteria used in practice are based on lists of test goals which need to be satisfied. The test goals themselves are typically either state or path predicates. This prototypical setting is accounted for in the next definition.

Definition 8 ((State) Regular Coverage Criterion and Test Goals). *A* regular coverage criterion *Φ is a coverage criterion constructed in the following way:*

(i) There is a mapping $\Phi(\mathcal{T},A)$ which maps $\mathcal{T}$ and A to a list of test goals *$\Phi(\mathcal{T},A) = \{\Psi_1,\dots,\Psi_k\}$.*
(ii) This mapping induces the coverage criterion $\Phi_A^{\mathcal{T}}$ as follows:

$$\Gamma \models \Phi_A^{\mathcal{T}} \ \textit{iff} \ \bigwedge_{i=1}^{k} \Pi_A^{\mathcal{T}} \models \Psi_i \Rightarrow \Gamma \models \Psi_i$$

Intuitively, this amounts to the following coverage criterion: "For each test goal which is feasible in $\Pi_A^{\mathcal{T}}$, the test suite Γ must contain a concrete test case."
Φ is a state regular coverage criterion*, if $\Phi(\mathcal{T},A)$ contains only state predicates.*

As an example, consider basic block coverage $\mathsf{BB}_A^{\mathcal{T}}$, which is a state regular coverage criterion: induced by the test goals $\mathsf{BB}(\mathcal{T},A) = \{\mathsf{block}_1,\dots,\mathsf{block}_k\}$. Here k denotes the number of basic blocks in $\mathcal{T}$, and each predicate block_i holds true at the first statement of the i-th basic block in the program.

We will now define test goal automata which are used to specify the test goals needed in regular coverage criteria.

Definition 9 (Test Goal Automaton). *A* test goal automaton *Q is a trace automaton where each accepting state a gives rise to a test goal Ψ_a:*

$$\pi \models \Psi_a \ \textit{iff} \ \exists t.\pi \models t \textit{ and } Q \textit{ accepts } t \textit{ in state } a$$

Thus, the test goal Ψ_a requires a path matched by a trace which Q accepts in state a. The test goal automaton Q naturally induces a regular coverage criterion $\mathsf{cov}[Q]$ based on the set $\mathsf{cov}[Q](\mathcal{T},A) = \{\Psi_a \mid a \in \mathsf{acc}(Q)\}$ of test goals.

Note that a single path may match more than one test goal simultaneously: First, each path is matched by a number of different traces, and second, more than one accepting state may be reached through a trace in Q.

We conclude this section with a formal definition of program queries, as introduced in Section 1.

Definition 10 (Program Query & Result). *A* program query $\langle A, Q\rangle$ *consists of an observation automaton A and a test goal automaton Q. A valid* result *to the query* $\langle A, Q\rangle$ *on transition system* $\mathcal{T}$ *is a test suite* $\Gamma \subseteq \Pi_A^{\mathcal{T}}$ *with* $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$.

3 Syntax and Semantics of FQL

The FSHELL query language FQL facilitates the specification of test suites over C source code. To decouple the language from the algorithmic details of the query engine, and to provide leeway for different query solving backends, we designed FSHELL as a declarative language. FQL contains three layers which reflect the formal model of Section 2:

(i) state predicates over program variables and the program counter,
(ii) trace automata to express both observation automata and test goal automata, and
(iii) program queries to express coverage criteria.

In the following subsections, we will describe these layers along with examples referring to Listing 1. Due to length restrictions, the presentation of FQL is kept informal; we refer the reader to [11] for more details. Section 4 describes our query solving engine based on bounded model checking.

3.1 State Predicates

We have seen in Section 2 that sets of state predicates are at the center of our formal model. For instance, basic block coverage is induced by the set of test goals $\mathsf{BB}(\mathcal{T}, A) = \{\mathsf{block}_1, \ldots, \mathsf{block}_k\}$. FQL is therefore equipped with *predicate generators* to extract sets of predicates from the C source code, and to create new sets of predicates. For example, the predicate generator `@blocks` yields the set $\{\mathsf{block}_1, \ldots, \mathsf{block}_k\}$ of predicates. Note that each block_i has the form $\mathsf{pc} = \text{const}$ where pc is the program counter. Syntactically, all predicate generators are prefixed with "`@`". Semantically, a predicate generator either yields a set of predicates over the program counter pc, or a set of predicates over the program variables.

Many predicate generators are used to extract sets of predicates from the source code. Examples of such predicate generators include `@file(bla.c)` which captures all program counter values of statements in the source file `bla.c`, `@func(main)` which captures the statements in function `main`, `@line(3)` to capture the statements in line 3, `@call(cmp)` to match all function calls of `cmp`, and `@entry` as well as `@exit` which capture all function entry and exit points respectively. In case of Listing 1 we get, e.g., `@call(cmp)` $= \{\mathsf{pc} = 13, \mathsf{pc} = 15, \mathsf{pc} = 17\}$.

To introduce new predicates not present in the source code, we use the predicate generator `@new-pred(cond)`, where cond is an arbitrary side-effect free C expression. For example, `@new-pred(x <= 7)` generates a singleton set $\{x \leq 7\}$ of state predicates.

For certain coverage criteria such as MC/DC, we also need the predicate generator `@grouped-conditions` which generates a *set of sets*, where each inner set captures the program counter values of the individual predicates which constitute a decision. Returning to Listing 1, we have `@grouped-conditions` $= \{\{\mathsf{pc} = 2\}, \{\mathsf{pc} = 4\}, \{\mathsf{pc} = 19, \mathsf{pc} = 20, \mathsf{pc} = 21\}\}$. To support the succinct formulation of most relevant coverage criteria, FQL contains a rich variety of predicate generators and can be easily extended with further ones without conceptual changes to the language [11].

Operations on Sets of State Predicates. Given two sets A and B of state predicates, FQL provides the following set-theoretic operations:

(and)	$A\&B \equiv \{a \wedge b \mid a \in A, b \in B\}$	$A, B \equiv A \cup B$	(union)
(or)	$A\|B \equiv \{a \vee b \mid a \in A, b \in B\}$	$A \backslash B \equiv A \setminus B$	(difference)
(negation)	$!A \equiv \{\neg a \mid a \in A\}$	$2\hat{\ }A \equiv \{A' \| A' \subseteq A\}$	(powerset)

We add `big-and(`A`)` $\equiv \bigwedge_{a \in A} a$ and `big-or(`A`)` $\equiv \bigvee_{a \in A} a$ to describe the conjunction and disjunction of all elements of a set of state predicates. To apply an operation to each element in a set, or to *each set in a set of sets*, we introduce the `set()` operator. Moreover, `union()` forms a single set from a set of sets. Given a set S of sets and an operation $o(s)$ on a set s of state predicates, we define:

$$\texttt{set(}o(s)\texttt{ : }s\texttt{ in }S\texttt{)} \equiv \{o(s) \mid s \in S\} \qquad \texttt{union(}S\texttt{)} \equiv \bigcup_{s \in S} s$$

Operations on Conditions. In describing coverage criteria, *conditions* occurring in the source code play a crucial role. A condition is an atomic expression which is possibly combined with other conditions using &&, ||, and ! to compute the decision involved in executing an **if**, **for**, **while**, **switch** or ?: statement. The generator `@pred-wo-loc()` extracts conditions from source code locations identified by program counter values. In addition, `@predicate()` and `@neg-predicate()` conjoin the extracted conditions with the corresponding predicate over the program counter. For example, let $C = \{\mathsf{pc} = 19, \mathsf{pc} = 20, \mathsf{pc} = 21\}$ be such a set, referring to Listing 1. Then we have

$$\begin{aligned}\texttt{@pred-wo-loc(}C\texttt{)} &= \{xy = 1, yz = 1, xz \neq 1\} \\ \texttt{@predicate(}C\texttt{)} &= \{\mathsf{pc} = 22 \wedge xy = 1, \mathsf{pc} = 22 \wedge yz = 1, \mathsf{pc} = 22 \wedge xz \neq 1\} \\ \texttt{@neg-predicate(}C\texttt{)} &= \{\mathsf{pc} = 22 \wedge xy \neq 1, \mathsf{pc} = 22 \wedge yz \neq 1, \mathsf{pc} = 22 \wedge xz = 1\}\end{aligned}$$

Note that $\mathsf{pc} = 22$ refers to the location of the decision inside which the conditions in C occur.

State Regular Coverage Criteria. Besides simple test goals such as `@blocks`, FQL can can also describe more complex coverage criteria. We illustrate this feature on the example of *multiple condition coverage*. Recall that multiple condition coverage requires

a test suite to cover—for each decision—all Boolean combinations of all conditions occurring in the respective decision. The test goals are therefore given by

```
union( set(
 union( set( big-and(@predicate(I) & @neg-predicate(D\I)) : I in 2^D ) ) :
 D in @grouped-conditions ) )
```

Hierarchical Navigation. In practical queries, the predicate generators `@file(bla.c)`, `@line(3)`, `@func(foo)`, as well as `@entry` and `@exit` occur quite frequently. We therefore allow the following abbreviations which facilitate hierarchical navigation in the source code:

```
/bla.c/ = @file(bla.c)          /bla.c/42 = @file(bla.c) & @line(42)
   foo/ = @func(foo)        /bla.c/foo/ = @file(bla.c) & @func(foo)
  foo/^ = @entry(foo)             foo/$ = @exit(foo)
 foo/SP = @func(foo) & SP      /bla.c/SP = @file(bla.c) & SP
```

In the last line, *SP* is to be replaced by any state predicate expression. Note that FQL also supports macros for frequently used expressions such as complex coverage criteria. Due to space restrictions we do not describe the macro feature in detail.

3.2 Trace Automata

Recall that trace automata are used to define path predicates, and to act as both observation automata and test goal automata. By implicit existential quantification, every state predicate can also be viewed as a path predicate, and it is easy to construct the corresponding automaton. Moreover, a set of state predicates naturally gives rise to an automaton with one accepting state for each state predicate in the set. For example, `@blocks` corresponds to an automaton with |`@blocks`| accepting states, one for each basic block. The following list exemplifies the most important automata theoretic operations of FQL which enable the user to manipulate and combine trace automata explicitly: Let A_1, A_2, A_3 be trace automata:

$$A_1\texttt{,}A_2 \equiv A_1 \cup A_2 \qquad \text{(union)}$$
$$A_1\texttt{->}A_2 \equiv A_1 \circ \mathsf{true}^* \circ A_2 \qquad \text{(sequencing)}$$
$$A_1\texttt{-[}A_3\texttt{]>}A_2 \equiv A_1 \circ A_3^* \circ A_2 \qquad \text{(restricted sequencing)}$$

Consider for example `main/^->main/$` over Listing 1: the traces of this automaton will match those program executions which pass the exit of `main` (line 19). In contrast, `main/^-[@file(bla.c)\@label(ERROR)]>main/$` requires that between the entry and the exit of `main` only locations other than those labeled “ERROR” (line 18) are seen. Note that each of these operations corresponds to a specific automata theoretic construction. Due to the special role of accepting states in defining test goals, we cannot use the standard automata theoretic minimization techniques, cf. [11].

3.3 Program Queries

We are now ready to define the program queries introduced in Section 1. Let A and B be FQL expressions which can be interpreted as trace automata (i.e., either trace automata, or sets of predicates as explained in the previous section). Then **cover** Q **passing** A expresses the program query $\langle A, Q\rangle$ with the semantics given in Definition 10.

Recall from Section 1, that FQL queries can also have a prefix. This prefix restricts all state predicates to a certain program part, e.g., a certain file. It is easy to see that the prefix can be moved into A and Q. For example, a query such as

```
> in /bla.c/ cover @line(4),@call(cmp)
  passing @file(bla.c)\@call(not_implemented)
```

which states that both, line 4 and a function call to `cmp` in file `bla.c` must be covered without ever calling `not_implemented()`, is equivalent to

```
> cover /bla.c/4,/bla.c/@call(cmp)
  passing @file(bla.c)\@call(not_implemented)
```

4 Query Processing Algorithms

In this section we describe the query processing algorithms. We first outline how program source code and a query are mapped to a SAT instance, and then detail on iterative and groupwise constraint strengthening in Section 4.2.

4.1 Program Instrumentation and Interfacing with CBMC

Bounded model checkers such as CBMC reduce questions about program paths to Boolean constraints in conjunctive normal form (CNF) which are solved by standard SAT solvers. Our query solving algorithms ICS and GCS employ the functionality of CBMC to obtain SAT instances suitable for test case generation. Recall that on input of a program annotated with assertions, CBMC outputs a SAT instance whose solutions describe program paths leading to assertion violations. To make this functionality useful for test case generation, we first instrument the program with the observation automaton A such that the resulting program reaches a failing assertion in the course of an execution, iff this program execution is matched by A. We therefore implement A as a C function that *monitors* program execution. To this end, the program $\mathcal{P}$ is instrumented to contain a *logging* layer, which reports the matching predicates after each executed step to the monitor. Moreover, we inject the test goal automaton as a second monitor, which only keeps track of the states of the test goal automaton in a distinguished variable, but does not cause assertion violations. Then, using CBMC, the instrumented program is transformed into the CNF-formula $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ which is satisfied by all program executions which reach an accepting state of A within a bounded number of steps. By construction, $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ contains distinguished Boolean variables referring to the state of the query automaton Q; these variables can be used to express the individual test goals. Therefore, a constraint of the form $\phi[\pi \in \Pi_A^{\mathcal{T}}] \wedge \phi[a]$ will satisfy those program executions which (i) respect observation automaton A and (ii) satisfy test goal Ψ_a. *In the rest of this section, we will for simplicity write this constraint as* $\pi \in \Pi_A^{\mathcal{T}} \wedge \pi \models \Psi_a$*, and tacitly assume the translation to CBMC described above.*

4.2 Guided SAT Enumeration

To generate a test suite Γ for a transition system $\mathcal{T}$ matching the query $\langle A, Q \rangle$, i.e., to achieve $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$, we introduce *iterative constraint strengthening (ICS).* In ICS, we build a test suite Γ iteratively from a sequence of test suites $\Gamma_0 \subset \Gamma_1 \subset \cdots \subset \Gamma_m$ with $\Gamma_0 = \emptyset$ and $\Gamma_q = \{\pi_1, \ldots, \pi_q\}$ for $1 \leq q \leq m$. In the m-th iteration, we reach a fixpoint when no more new goals can be covered.

Algorithm Overview. In the q-th iteration we build the *path constraint* ICSPC_q (Equation (1)) and obtain the test case π_{q+1} as one of its solutions. Here, ICSPC_q describes those paths in $\Pi_A^{\mathcal{T}}$ which cover a hitherto uncovered test goal. If no such test goal exists any more, ICSPC_q becomes unsatisfiable. Having determined a new test case π_{q+1}, we build ICSPC_{q+1} and continue the procedure with the $(q+1)$-st iteration until we reach an iteration m where ICSPC_m becomes unsatisfiable.

In order to fit the framework of *incremental SAT solving* (cf. [12]), we rewrite ICSPC_q (Equation (2)) in such a way that we are able to describe ICSPC_{q+1} incrementally in terms of ICSPC_q by *only adding* new constraints *without removing or changing* previously added constraints (Equation (3)). Using this incremental formulation of ICSPC_q, we describe iterative constraint strengthening (ICS) based upon an incremental SAT solver in Listing 2. The m paths finally collected by ICS constitute indeed a covering test suite (Theorem 1).

Path Constraints. The initial path constraint ICSPC_0 requires that a path is in $\Pi_A^{\mathcal{T}}$ and covers at least one of the test goals Ψ_a for $a \in \mathsf{acc}(Q)$. Subsequently, in ICSPC_q, we require the path to cover at least one test goal Ψ_a which remained *uncovered* by the test suite Γ_q. Since Γ_{q+1} must cover at least one more test goal than Γ_q, it suffices to *strengthen* the constraint ICSPC_q to obtain ICSPC_{q+1}. Below, we write $\mathsf{uncov}_q = \{a \in \mathsf{acc}(Q) \mid \Gamma_q \not\models \Psi_a\}$ for the set of accepting states which correspond to test goals not covered in Γ_q. Note that $\mathsf{uncov}_0 = \mathsf{acc}(Q)$ since $\Gamma_0 = \emptyset$ covers no test goals at all. Then, for $0 \leq q \leq m$, we search for a solution π_{q+1} to the q-th constraint

$$\mathsf{ICSPC}_q(\pi) := \pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{a \in \mathsf{uncov}_q} \pi \models \Psi_a \tag{1}$$

Note that the empty disjunction is equivalent to false, i.e., if $\mathsf{uncov}_q = \emptyset$, then $\mathsf{ICSPC}_q \equiv \mathsf{false}$. Thus, ICSPC_q is satisfied by exactly those paths in $\Pi_A^{\mathcal{T}}$ which satisfy at least one *feasible* test goal still *uncovered* by Γ_q. If no such test goal exists, i.e., if Γ_q *achieves coverage,* then ICSPC_q is unsatisfiable.

Incremental Path Constraints. In incremental SAT solving, we use a single persistent clause database for consecutive solver invocations. When the SAT solver finds a solution, we add new clauses to the clause database, but do not remove any clauses. When the execution of the SAT solver is continued, the learned clauses obtained during earlier invocations remain valid and help to guide the search of the solver. Therefore, we have to construct ICSPC_{q+1} from ICSPC_q by only adding further constraints to the clause database. Observe that $\mathsf{uncov}_{q+1} \subset \mathsf{uncov}_q$ holds for $0 \leq q \leq m-1$. Thus in going from ICSPC_q to ICSPC_{q+1}, we have to remove all test goals Ψ_a with $a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$

from the disjunction $\bigvee_{a \in \mathsf{uncov}_q} \pi \models \Psi_a$ occurring in Equation (1). To do so, we introduce a new Boolean variable S_a for each accepting state $a \in \mathsf{acc}(Q)$ and write ICSPC_q equisatisfiable as

$$\mathsf{ICSPC}_q(\pi) := \left[\pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{a \in \mathsf{acc}(Q)} (S_a \wedge \pi \models \Psi_a) \right] \wedge \bigwedge_{a \notin \mathsf{uncov}_q} \neg S_a \tag{2}$$

Thus ICSPC_q consists of (a) an initial expression, shown above in square brackets, which remains unchanged throughout all iterations, and (b) a conjunction which is expanded from one iteration to the next. Adding $\neg S_a$ to the constraint renders the corresponding disjunct $S_a \wedge \pi \models \Psi_a$ unsatisfiable, and therefore only the disjuncts for $a \in \mathsf{uncov}_q$ remain enabled. Note that for ICSPC_0 we have $\mathsf{true} \equiv \bigwedge_{a \notin \mathsf{uncov}_0} \neg S_a$. Thus, in each iteration step, we use

$$\mathsf{ICSPC}_{q+1}(\pi) := \mathsf{ICSPC}_q(\pi) \wedge \bigwedge_{a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}} \neg S_a \tag{3}$$

to obtain ICSPC_{q+1} from ICSPC_q. Since we only add further constraints conjunctively, this approach fits the requirements of incremental SAT solving.

Iterative Constraint Strengthening. In our presentation of the algorithm, we assume a SAT solver which supports the following methods: (a) *Adding constraints* with add(ϕ): The method takes an arbitrary constraint ϕ over variables from arbitrary finite domains. While we use such a general interface to simplify the presentation of our algorithm, our implementation is based upon the SAT instance $\phi[\pi \in \Pi_A^{\mathcal{T}}]$ which we described in Section 4.1. (b) *Checking for satisfiability* with satisfiable(): The method returns true iff there exists a solution to the constraints added to the clause database so far. If a call to satisfiable() returns true, a witness is cached. (c) *Obtaining a solution* with solution(): The method returns the last witness cached in a call to satisfiable().

Listing 2. Iterative Constraint Strengthening (ICS)

```
1  func ICS(Π_A^T, ⟨A,Q⟩)
2  begin
3    q := 0; Γ_0 := ∅; uncov_0 := acc(Q);
4    add(π ∈ Π_A^T ∧ ⋁_{a∈acc(Q)} (S_a ∧ π ⊨ Ψ_a));
5    while satisfiable () do begin
6      π_{q+1} := solution();
7      Γ_{q+1} := Γ_q ∪ {π_{q+1}}; uncov_{q+1} := ∅;
8      forall a ∈ uncov_q do
9        if π_{q+1} ⊨ Ψ_a then add(¬S_a);
10       else uncov_{q+1} := uncov_{q+1} ∪ {a};
11     q := q+1;
12   end;
13   return Γ_q;
14 end;
```

The resulting procedure ICS is shown in Listing 2. In line 3 we initialize the iteration counter q, the first test suite Γ_0, and the set of test goals uncov_0 uncovered by Γ_0. Then in line 4, we add the initial expression from Equation (2) and start the search for the first solution in line 5. If a solution is found, it is obtained from the solver, assigned to π_{q+1}, and added to Γ_{q+1}. Then, after initializing uncov_{q+1}, we update the clause database following Equation (3) and fill the set uncov_{q+1} in lines 8 to 10: For each yet uncovered state $a \in \mathsf{uncov}_q$, we check whether π_{q+1} satisfies Ψ_a. If this is the case, $a \in \mathsf{uncov}_q \setminus \mathsf{uncov}_{q+1}$ holds, and thus we add $\neg S_a$ in line 9. Otherwise a remains uncovered by Γ_{q+1} and hence we add a to uncov_{q+1} in line 10. Once no further solution is found in line 5, the accumulated suite Γ_q is returned.

Theorem 1 (Correctness of Iterative Constraint Strengthening). *The test suite* Γ *returned by the algorithm* $\mathrm{ICS}(\Pi_A^{\mathcal{T}}, \langle A, Q \rangle)$ in Listing 2 satisfies $\Gamma \models \mathsf{cov}[Q]_A^{\mathcal{T}}$.

Remark 3 (Nondeterminism in Choosing π_{q+1}). Our algorithm leaves the particular choice of π_{q+1} open to the underlying SAT solver (line 6). Potential optimizations could control this choice to minimize the number of test cases necessary to obtain coverage.

Groupwise Constraint Strengthening. Certain regular coverage criteria, such as predicate complete or multiple condition coverage, require an *exponential number of test goals*. For example, recall that multiple condition coverage (Section 3.1) has one test goal for each basic block and *each possible evaluation of all conditions* involved in deciding which edge to choose in leaving the basic block. Hence, the number of test goals is exponential in the number of conditions in each decision. For this reason, the disjunction in ICSPC_0 will be of exponential size—thus rendering iterative constraint strengthening hard for such coverage criteria.

To mitigate this situation, we introduce *groupwise constraint strengthening (GCS)* as an optimization of iterative constraint strengthening. GCS can be combined with ICS and allows to handle all test goals which are state predicates. Let us thus for simplicity assume that all test goals Ψ_a for $a \in \mathsf{acc}(Q)$ are state predicates. To apply GCS, we require the test goals to be partitioned into k distinct *groups* $G_i = \{\Psi_i^1, \ldots, \Psi_i^{k_i}\}$ *of mutually exclusive test goals* for $1 \leq i \leq k$, i.e., we require that there exists no *state* s with $s \models \Psi_i^g$ and $s \models \Psi_i^h$ for all $1 \leq g \neq h \leq k_i$ and $1 \leq i \leq k$.

In the GCS algorithm, we avoid the construction of the initial and very large disjunction $\bigvee_{a \in \mathsf{uncov}_q} \pi \models \Psi_a$, as it appears in ICSPC_q (Equations (1) and (2)): Instead of individual test goals, we use a small number of *compound test goals* comp_i, where each compound test goal represents the goals of the whole group $G_i = \{\Psi_i^1, \ldots, \Psi_i^{k_i}\}$ of individual test goals Ψ_i^j. To represent group G_i, its compound test goal comp_i has to be semantically equivalent (but usually not identical) to $\bigvee_{j=1}^{k_i} \Psi_i^j$. It is important to note however that in many practical cases, comp_i can be *formulated much more succinctly* than $\bigvee_{j=1}^{k_i} \Psi_i^j$. For example, in case of multiple condition coverage, we partition the goals into groups according to the blocks they relate to. Then, $s \models \mathsf{comp}_i$ holds for a state s iff s visits the i-th basic block, i.e., comp_i has the form $\mathsf{pc} = \mathrm{const}$.

Starting with the compound test goal comp_i, we add for each covered test goal Ψ_i^j of group G_i, i.e., for each $\Psi_i^j \in G_i \setminus \mathsf{uncov}_q$, its negation $\neg\Psi_i^j$ to the corresponding compound test goal. This approach yields for each group G_i an *aggregate test goal*

$$\mathsf{aggr}_i^q := \mathsf{comp}_i \wedge \bigwedge_{\Psi_i^j \in G_i \setminus \mathsf{uncov}_q} \neg\Psi_i^j \tag{4}$$

Since we use aggr_i^q to represent the remaining uncovered test goals $G_i \cap \mathsf{uncov}_q$ of the group G_i in iteration q, we will rely on the equivalence

$$\mathsf{aggr}_i^q \equiv \bigvee_{\Psi_i^j \in G_i \cap \mathsf{uncov}_q} \Psi_i^j \tag{5}$$

which follows from the construction and the mutual exclusiveness of the test goals within each group G_i. Written in the form of Equation (5), aggr_i^q does not explicitly

refer to any infeasible test goals and only involves *feasible* test goals as subexpressions. This significantly reduces the size of the constructed constraint.

Having defined aggr_i^q in this way, GCS proceeds like ICS but with Equation (1) replaced by

$$\mathsf{GCSPC}_q(\pi) := \pi \in \Pi_A^{\mathcal{T}} \wedge \bigvee_{i=1}^{k} \pi \models \mathsf{aggr}_i^q \tag{6}$$

Similar to ICS we also adopt GCSPC_q to fit incremental SAT solving: More precisely, we leave the overall constraint (Equation (6)) unchanged and replace aggr_i^q (Equation (4)) by an equisatisfiable and incrementally expandable expression. Thus, we can incrementally strengthen aggr_i^q for each group individually.

The effectiveness of GCS as an optimization of ICS relies on three conditions: (a) The overall number of groups must be small, since we maintain for each group G_i a constraint aggr_i^q. (b) The compound test goal comp_i must be available in a succinct formulation. (c) The fraction of *feasible* test goals Ψ_i^j in each group G_i must be small, since the negation of each feasible test goal is added to aggr_i^q in some iteration q. Conditions (a) and (b) hold for important coverage criteria such as multiple decision or predicates complete coverage. If condition (c) does not hold, then the number of required test cases will be large – but this is inherent in the coverage criterion and not an artefact of GCS.

Remark 4 (Mutual Exclusiveness: State vs. Path Predicates). It is tempting to assume that the mutual exclusiveness defined in terms of states is easily generalized to the level of path predicates. However, this is not the case as mutually exclusive state predicates *do not result* in mutually exclusive path predicates because of their implicit existential quantification, cf. Definition 3 and Remark 1.

5 Experimental Results

In our experiments we investigated test case generation for basic block (BB) and condition coverage (CC). We performed our experiments on a 3.0 GHz AMD64 system with 8 GB RAM. The table below summarizes our results with respect to BLAST. The column "Min" shows the number of test cases removed by our test suite minimization algorithm. Our current implementation of FSHELL is an optimized version of that presented in [4]. It generates fewer test cases, and, after test case generation for basic block coverage,

		BLAST (BB)		BB			CC	
Source file	LLOC	#cases	Time[s]	#cases	Time[s]	Min*	#cases	Time[s]
`kbfiltr.i`	4879	39	300	26	18	6	98	24
`floppy.i`	6435	111	1500	63	1041	10	175	1259
`cdaudio.i`	8022	85	1500	71	1240	7	161	1243
`parport.i`	20698	213	5460	134	1859	21	351	2915
`parclass.i`	45283	219	2520	156	1324	16	392	2070
`matlab.c`	2033	-	-	5	30	1	16	31
`autopilot.i`	3141	-	-	206	894	14	450	1358

FSHELL minimizes an obtained test suite. The results for BLAST are taken literally from [3], because the version of BLAST performing test case generation is currently unavailable. Beyer et al. performed their experiments on a 3.06 GHz Dell Precision 650 with 4 GB RAM. FSHELL outperforms BLAST, as we achieve coverage with fewer test cases faster. Besides the experiments on the device drivers from BLAST we conducted experiments on an engine controller (`matlab.c`) provided by an industrial collaborator from the automotive industries. It is generated from a MATLAB/Simulink model. Furthermore, we ran our tool on preprocessed sources (`autopilot.i`) generated from source code in PapaBench[2]. The results show that FSHELL scales well when moving from basic block coverage to condition coverage. Experiments concerning more sources and more complex queries can be found in [11].

6 Conclusion

In this paper, we introduced a query language for test case specification together with a query solving backend based on bounded model checking. Our backend is based on two new algorithms which guide the SAT solver to efficiently enumerate a test suite. Our implementation FSHELL demonstrates the effectiveness and versatility of our approach.

References

1. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO. (2004) 1–22
2. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. (2004) 168–176
3. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: ICSE. (2004) 326–335
4. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In: CAV. (2008) 209–213
5. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: SPIN. (2003) 235–239
6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast Query Language for Software Verification. In: SAS. (2004) 2–18
7. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: TACAS. (2002) 327–341
8. Tan, L., Sokolsky, O., Lee, I.: Specification-based testing with linear temporal logic. In: IRI. (2004) 493–498
9. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: ISSTA. (2004) 97–107
10. Hamon, G., de Moura, L.M., Rushby, J.M.: Generating Efficient Test Sets with a Model Checker. In: SEFM. (2004) 261–270
11. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. Technical Report TUD-CS-2008-1013, TU Darmstadt (2008)
12. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT. (2003) 502–518

[2] `http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`

Type-Safe Bytecode Generation in Scala

Peter Thiemann
Universität Freiburg

Mnemonics is a novel bytecode generation library for the Java Virtual Machine (JVM), written in Scala. It avoids the generation of ill-formed bytecode sequences by adopting a typed representation of the stack and the local variables. Individual instructions are modeled as functions that transform these types according to the action of the instruction. The library exploits a number of advanced features of Scala's type system (type inference with bounded polymorphism, objects with type members, implicit parameters, and reflection) to guarantee that the compiler rejects illegal combinations of instructions at compile time.

Programmierung als Leitbild in der Theorie der Ökosysteme

Baltasar Trancón y Widemann
Universität Bayreuth

Die Umweltwissenschaften haben im Lauf ihrer Entstehung als Disziplin das Forschungsprogramm der klassischen Physik mit dem Paradigma der mechanistischen Systeme weitgehend unhinterfragt übernommen. Es zeigt sich, dass dieses Vorgehen umso problematischer ist, je mehr lebende Bestandteile das Verhalten des Systems dominieren. Andererseits gibt es zahlreiche alte, gesellschaftlich hochrelevante und erfolgreiche Nutzungstraditionen von Umweltsystemen, die theoriefrei und rein heuristisch funktionieren. Als Folge entsteht eine im Bereich der Naturwissenschaften einmalige Spannung zwischen Theorie und Praxis, und es zeigen sich Symptome einer wissenschaftlichen Krise, wie sie oft nur durch einen Paradigmenwechsel zu lösen ist. Dieser Beitrag soll dreierlei zeigen: Erstens, dass aktuelle Entwicklungen im Bereich der computergestützten Umweltmodellierung sich dem Informatiker als Krisensymptome darstellen, die dem „laienhaften Anwender“verschlossen bleiben. Zweitens, dass Begriffe aus der Softwaretechnik und der Programmanalyse geeignet sind, die Rolle von Theorie und Modellierung von Ökosystemen metaphorisch zu beschreiben. Drittens, dass sich Formalismen aus dem Bereich der Semantik von Programmiersprachen und -kalkülen ganz konkret als Basis eines alternativen Paradigmas der Theorie der Ökosysteme eignen.

Programming by Equilibria

Christian Tschudin und Thomas Meyer
Universität Basel

We present a reactive computing paradigm that seeks to solve problems by bringing the programmed system into an equilibrium state: The production of a certain (numeric) result, or of a certain side effect (routing), emerges from the system's tendency to strive for an equilibrium. We have identified important equilibra for properties like adaptivity, as well as self-healing where a program controls its own code base. By linking our execution model with well known laws from chemistry, we can predict a program's dynamic behavior and in some cases even provide elegant proofs of convergence. In this paper we recapitulate how one can compute results out of the random execution of instructions; We then introduce an artificial chemistry as our reactive programming environment, in which some networking protocols can be expressed in a natural way. We present a gossip-style protocol for the cooperative computation of an average value which is amenable to a formal stability analysis and ultimately convergence proof.

Adding Weights to Dynamic Pushdown Networks

Extended Abstract

Alexander Wenner

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
alexander.wenner@uni-muenster.de

The interest in writing parallel programs has increased in recent years, especially with the emergence of programming languages with native support for parallelism and the increased availability of parallel hardware. However parallel programming is notoriously difficult and error-prone. Thus static analysis of parallel programs has become more and more important.

The goal of this talk is to present a generic framework for the analysis of parallel programs, especially in the presence of recursive procedures and dynamic process creation. We base our framework on dynamic pushdown networks (DPN) [1] and weighted pushdown systems (WPDS) [2].

DPN precisely model procedures and process creation, by modeling each process as a pushdown system (PDS), where rules can additionally create new PDS as a side effect. They have mainly been studied for reachability analyses [1, 3, 4]. Since the analysis of recursive procedures and synchronisation is undecidable [5], DPN do not model synchronisation between processes. However, through the addition of weights we will be able to analyse some interaction between processes.

WPDS extend PDS by labelling transitions with weights and solving the generalised pushdown predecessor (GPP) problem [2, 6, 7], which is the meet-over-all-paths solution for paths between regular sets of configurations. The weights can be used to formulate a wide range of analysis problems. The GPP problem formulation allows for a specific query depending on the shape of the entire call-stack, in contrast to standard dataflow techniques, where typically all information at the topmost program point is merged.

Analogous to WPDS we extend DPN to weighted DPN (WDPN) by annotating weights to transitions and study the corresponding GPP problem, which now allows for a query based on the state of each process in the network. Even though a WPDS is then simply a WDPN with one process, adapting the approach to solve the GPP problem from WPDS to WDPN is problematic. In general a path of a DPN is an interleaving of the transitions of arbitrary many parallel processes. Results from [1] show, that such a set of paths can not be described using a grammar as in [2] or a constraint system. Thus the solution of the GPP problem can not be computed as abstract interpretation [8] of such a system.

We avoid these problems by introducing a branching semantics for DPN, similar to the tree semantics in [4]. Transitions of newly spawned processes are

no longer mixed with the transitions of the creating process, but contained in their own branch. This results in executions which are tree shaped for single processes and form hedges, which contain a tree for each process, for configurations with multiple processes. The set of hedges connecting two regular sets of configurations can then be described using a constraint system, using an approach adapted from WPDS.

We introduce an extended weight domain to abstract these hedges, and study the analogous branching GPP (BGPP) problem, which is the meet-over-all-hedges solution, for these branching WDPN (BWDPN). We show, that if the weight domain of a WDPN and the extended weight domain of a BWDPN, both based on the same DPN, are related, the solution for the GPP problem of the WDPN can be derived from the solution of the corresponding BGPP problem of the BWDPN. The BGPP problem can be solved by abstract interpretation of the above mentioned constraint system.

Up to this point our framework of WDPN and BWDPN can solve the bitvector problems for DPN formulated in [1], the more general KILL/GEN analyses described in [9] and the shortest path analysis from [2]. In [10] a different approach to generalize WPDS to parallel programs is presented, by introducing a context bound. This leads to an underapproximation, whereas our approach handles unbounded context switches precisely.

References

1. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR. LNCS 3653, Springer (2005)
2. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comp. Prog. **58**(1-2) (2005)
3. Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability analysis of multithreaded software with asynchronous communication. In: FSTTCS. LNCS 3821, Springer (2005)
4. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: CAV. LNCS 5643, Springer (2009)
5. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. **22**(2) (2000)
6. Lal, A., Reps, T.W.: Improving pushdown system model checking. In: CAV. LNCS 4144, Springer (2006)
7. Lal, A., Reps, T.W., Balakrishnan, G.: Extended weighted pushdown systems. In: CAV. LNCS 3576, Springer (2005)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM Press (1977)
9. Lammich, P., Müller-Olm, M.: Precise fixpoint-based analysis of programs with thread-creation and procedures. In: CONCUR. LNCS 4703 (2007)
10. Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In: TACAS. LNCS 4963, Springer (2008)

Towards Dynamic Code Evolution for the HotSpot VM

Thomas Würthinger
Johannes Kepler Universität Linz

Dynamic code evolution is a technique to change the source code of a running program. The current hotswapping mechanism in the Java HotSpot VM only allows changing the bodies of methods at runtime. We are working on an approach that allows arbitrary changes. This talk is about our experiences implementing a prototype in HotSpot that supports adding and deleting of methods and fields as well as performing changes to the class hierarchy. What are the conditions for dynamic code evolution to have a clear semantics and what kind of changes should be forbidden? What are meaningful applications of dynamic code evolution for Java and their requirements? The talk will conclude with ideas for future work to make dynamic code evolution stable and performant.

Autorenverzeichnis

Altmeyer, Sebastian, 1
Amme, Wolfram, 13, 95
Ansaloni, Danilo, 51

Bajrović, Enes, 17
Barany, Gergö, 27
Berghammer, Rudolf, 38
Braßel, Bernd, 38
Binder, Walter, 51
Böszörmenyi, Laszlo, 178
Brandner, Florian, 65
Brunthaler, Stefan, 67
Burguiere, Claire, 1

Dörre, Jens, 68

Ertl, M. Anton, 69

Feller, Christoph, 76
Fischer, Sebastian, 77, 243

Grelck, Clemens, 78
Grund, Daniel, 1

Höger, Christoph, 104
Haas, Walter, 126
Hack, Sebastian, 93
Hanus, Michael, 94
Heinen, Jonathan, 117
Heinze, Thomas, 95
Herter, Jörg, 1
Holzer, Andreas, 269

Jansen, Christina, 117
Jebelean, Tudor, 124, 221

Kadlec, Albrecht, 155
Kirner, Raimund, 126, 141, 155
Knoop, Jens, 155

Lammich, Peter, 167
Lampl, Oliver, 178
Lucas, Philipp, 1

Maksoud, Abdel Mohamed, 1
Majchrzak, Tim A., 193
Mattick, Volker, 208
Mehofer, Eduard, 17
Meyer, Thomas, 286
Michel, Patrick, 219
Moret, Philippe, 51
Moser, Simon, 95

Neumerkel, Ulrich, 220

Parshin, Oleg, 1
Pister, Markus, 1
Popov, Nikolaj, 221
Prantl, Adrian, 155, 230
Puntigam, Franz, 231

Reck, Fabian, 243
Reineke, Jan, 1
Richter, Dirk, 141, 244

Schallhart, Christian, 269
Schlickling, Marc, 1
Scholz, Sven-Bodo, 78, 259
Scholz, Wolfgang, 260
Schordan, Markus, 155, 261
Schreiner, Dietmar, 262
Schwarz, Martin, 263
Shafarenko, Alex, 78
Stümpel, Annette, 265
Stadler, Lukas, 264

Tautschnig, Michael, 269
Thiemann, Peter, 284
Trancón y Widemann, Baltasar, 285
Tschudin, Christian, 286

Veith, Helmut, 269
Villazón, Alex, 51

Würthinger, Thomas, 289
Wachter, Björn, 1
Wenner Alexander, 287
Wilhelm, Reinhard, 1

Zimmermann, Wolf, 141